Computer Generation of Integral Images using Interpolative Shading Techniques

Graham E. Milnthorpe

A thesis submitted for the degree of Doctor of Philosophy

School of Engineering and Manufacture, De Montfort University, Leicester

December 2003

The following has been excluded at the request of the university

Pages 151 - 164

Research to produce artificial 3D images that duplicates the human stereovision has been ongoing for hundreds of years. What has taken millions of years to evolve in humans is proving elusive even for present day technological advancements. The difficulties are compounded when real-time generation is contemplated. The problem is one of depth. When perceiving the world around us it has been shown that the sense of depth is the result of many different factors. These can be described as monocular and binocular. Monocular depth cues include overlapping or occlusion, shading and shadows, texture etc. Another monocular cue is accommodation (and binocular to some extent) where the focal length of the crystalline lens is adjusted to view an image. The important binocular cues are convergence and parallax. Convergence allows the observer to judge distance by the difference in angle between the viewing axes of left and right eyes when both are focussing on a point. Parallax relates to the fact that each eye sees a slightly shifted view of the image. If a system can be produced that requires the observer to use all of these cues, as when viewing the real world, then the transition to and from viewing a 3D display will be seamless. However, for many 3D imaging techniques, which current work is primarily directed towards, this is not the case and raises a serious issue of viewer comfort. Researchers worldwide, in university and industry, are pursuing their approaches in the development of 3D systems, and physiological disturbances that can cause nausea in some observers will not be acceptable.

The ideal 3D system would require, as minimum, accurate depth reproduction, multi-viewer capability, and all-round seamless viewing. The necessity not to wear stereoscopic or polarising glasses would be ideal and lack of viewer fatigue essential. Finally, for whatever the use of the system, be it CAD, medical, scientific visualisation, remote inspection etc on the one hand, or consumer markets such as 3D video games and 3DTV on the other, the system has to be relatively inexpensive.

Integral photography is a 'real camera' system that attempts to comply with this ideal; it was invented in 1908 but due to technological reasons was not capable of being a useful autostereoscopic system. However, more recently, along with advances in

technology, it is becoming a more attractive proposition for those interested in developing a suitable system for 3DTV.

The fast computer generation of integral images is the subject of this thesis; the adjective 'fast' being used to distinguish it from the much slower technique of ray tracing integral images. These two techniques are the standard in monoscopic computer graphics whereby ray tracing generates photo-realistic images and the fast forward geometric approach that uses interpolative shading techniques is the method used for real-time generation. Before this present work began it was not known if it was possible to create volumetric integral images using a similar fast approach as that employed by standard computer graphics, but it soon became apparent that it would be successful and hence a valuable contribution in this area. Presented herein is a full description of the development of two derived methods for producing rendered integral image animations using interpolative shading. The main body of the work is the development of code to put these methods into practice along with many observations and discoveries that the author came across during this task.

I would like to express my thanks to all the people in the 3D and Biomedical Imaging Group of De Montfort University for the support they have given me during this research. Especial thanks to Malcolm McCormick for the encouragement he has given me and the trust he has shown in abilities that I did not realise I had. Thanks to Neil Davies for his constant determination for perfection in the computer generation of the integral images. Finally, thanks to Matt Forman for revealing to me the mysteries of Unix and the difficult job of keeping the systems up and running.

Funding for the research in chronological order came from The Defence and Research Agency (DERA), a contract (LAIRD) under the European Link/EPSRC photonics initiative, and DTI/EPSRC sponsorship within the PROMETHEUS project.

A	size of aperture
c	surface of curvature
d	distance of aperture to lens array
D	diameter of Fraunhoffer rings
f	focal length of lens
F, F^1	focal planes
H, H^1	unit planes
H	unit normal to a hypothetical surface that is oriented in a direction half-way between the light direction vector and the viewing vector
I_i	intensity of a point light source
I_r, I_g, I_b	colour components of an intensity
k_a	ambient coefficient
k_d	wavelength-dependent empirical reflection coefficient
k_s	specular coefficient
kl	lenslet number reading from the bottom of the array
ka	projection point number reading from the top of the proj. point plane
kp	pixel number relative to the pixel number zero set at centre of array
L	light direction vector
L,M,N	direction cosines
max_zv	maximum z-coordinates of the scene
min_zv	minimum z-coordinates of the scene
N	normal to a surface
N	number of lenslets
n	number of pixels behind each lenslet
n	index that simulates surface roughness
n_1	refractive index of air and is taken to be the value of 1
n_2	refractive index of lens material
P	image point
P	plenoptic function
P	pitch of lenslet
$\mathbf{R_l}$	refraction matrix
R	specular direction
r	radius of curvature of a lens
S	system matrix
s	spacing between points making a line in 3D space
$\mathbf{T_l}$	translation matrix
t	time
t^1	thickness of a lenslet
V_1, V_2, V_3	view vectors
V_x, V_y, V_z	view coordinates
$\mathbf{V_a}, \mathbf{V_b}$	vectors from each triangle to the 2 aperture extremes
x	variable multiplier of distance d
x,y,z	Cartesian coordinates
x,y	pixel coordinates
x1,y1,z1	start point coordinates

x2,y2,z2	end point coordinates
xr, yr	distances of the image plane hit from the lenslet 's vertex
*x_fin, y_*fin	image plane coordinates
Xo,Yo,Zo	object point coordinates
Xi,Yi,Zi	image point coordinates
zv_pos	a fraction of the whole scene depth
Zv,zv	z-coordinate position of the lens array
Zap	z-coordinate position of the aperture
α, β	angles of seperation from the x and y axis
λ	wavelength
θ,σ,ϕ	angles
θ_1	the incident angle of a ray to a lens
θ_2	the refracted angle of a ray from a lens

1

Introduction

1.1 Discussion

3D imaging in the context of this thesis is the displaying of a spatial image that has real volume and not the 3D imaging that is a 'catchword' for the navigable representation of the real world on a 2D display. The latter was originally justified because of the change from the vector wire frame image to the new raster graphics that produced more photographic quality images. These included depth cues that were not possible with wire frame images, for example perspective, interposition and shading which aids in the visualization process to make the images seem "real". These cues, though, are entirely monocular, or single-eye, depth cues.

Ideally, to replicate a 3D scene volumetrically, the optimum recorded intensity distribution of a scene must contain within it the intensity of the light rays from every possible angle and wavelength, intersecting an infinite number of points covering a capture medium. Then potentially all possible depth cues, monocular and binocular will be present. Simultaneously to extract all of this recorded information, in such a way as to spatially replay an infinite number of views of the original scene, is the task of volumetric 3D imaging. This is equivalent to making a perfect spatial copy of the original scene, and is the ultimate goal. However, the best that can reasonably be achieved today is a less rigorous approach and information reduction is a key factor in the scenario. This can be seen, for example, when the display mechanism is pixelated and the rays are parameterised in terms of (x,y) pixel coordinates. As the parameterisation moves towards infinity (or non-parameterisation) and the pixel size becomes smaller, more direct discrete information can be captured and replayed. The closer systems approximate to this ultimate goal the more exact is the imitation of the original scene.

The ultimate goal can be expressed as capturing and replaying the plenoptic function [1]. The plenoptic function is a complete holographic representation of the visual

world that implicitly contains a description of every possible photograph that could be taken of a particular space-time chunk of the world. To measure the plenoptic function it is imagined that an idealized eye is placed at every possible location (V_1,V_2,V_3) and the intensity of the light rays passing through the centre of the pupil for each angle (θ,σ), each wavelength (λ) at each time (t) is recorded. As long as the eye always looks in the same direction the angles are computed with respect to an optical axis that is parallel to the V_z axis. The function can be expressed as:

$$P = P\ (\theta,\sigma,\lambda,\ t,Vx,Vy,Vz)$$

and with parameterisation:

$$P = P(x,y,\ \theta,\sigma,\lambda,t,Vx,Vy,Vz)$$

The 3D imaging system takes samples from this function and as the number of samples approaches infinity then so does the accuracy of the 3D space-time copy.

1.2 3D imaging systems

Table 1.1 is a brief analysis by category of some of the better-known 3D imaging systems presently developed. This introductory chapter is not a historical look at 3D imaging, the methods discussed here are those that are the more practical and have been more widely developed. For example, an ideal system would not require the viewer to wear glasses as in some stereoscopic displays and there is a concentrated effort presently in the 3D community to develop 'glasses free' autostereoscopic displays in the drive towards 3DTV. Consequently, this chapter focuses on autostereoscopic techniques that use lenticular or microlens arrays; namely multiview and integral imaging.

Stereoscopic	**anaglyphs**	Images are encoded in different colours, each eye sees the image through a different colour filter.	All require baffled viewing channels. Depth reproduction changes with viewing distance.
	polarisation	Images are encoded in orthogonal polarisation, each eye sees the image through an appropriate polarisation filter.	
	time sequential	Images are displayed sequentially, and a shutter over each eye opens in synchronisation with the display of the image for that eye.	
Autostereoscopic	**holography**	The reconstruction of wavefronts using phase information from the interface of a highly coherent source.	Coherent source of radiation must be used for recording. The subject must remain still during exposure.
	tracking systems	Correct information is channelled to the viewer's eyes using complex tracking systems.	Huge amount of processing to look after the tracking of the viewer. No good as multiviewer system.
	lenticular sheets	**Variety of systems use a physical sampling optical device to encode and decode the angular information contained in a scene. These include:**	
		multiview and stereo: The object is viewed by two or more cameras where each view is interdigitated behind a lenslet, and requires one pixel RGB set for each camera	The scene is discretised into planes and suffers from the effect of 'cardboarding'. Small angle of view and flipping between image fields. Possible eyestrain
		integral: A single scene is recorded from the numerous different viewing points of a lenticular sheet and replayed through a similar sheet	Optical models appear to exist in space hence provide monocular and binocular cues. Pseudoscopic reversal required. No viewer fatigue. Continuous parallax in all directions. Method is simple but effective though resolution requirements are higher than for multiview

Table 1.1: 3D imaging systems

Takanori Okoshi [2] analysed almost all 3D imaging methods ever devised and existing technologies, if not the same, are variations of those that are reported in his book. An amazing variety of patents, prototypes and papers can be found today on the subject of 3D imaging systems but in the area of integral imaging the papers and patents are few and far between. The majority of research in integral imaging for many years came from the 3D and Biomedical Imaging Group of De Montfort University (DMU) [3] and very recently with a great flourish from NHK Japan [4] and Seoul National University Korea [5]. The reasons for this original apathy are the complexities involved in reversing a pseudoscopic image and the difficulties associated with microlens manufacture. These problems have presented insurmountable obstacles at various times for 3D-imaging researchers. In computer graphics, however, pseudoscopy poses no problem and if omni-directional parallax microlenses have not yet reached sufficient quality for use in an integral imaging system then there is a plentiful supply of uni-directional or lenticular lens arrays. The same principles of integral imaging that apply to omni-directional parallax also apply to uni-directional parallax.

1.3 Directionality

When viewing a real-world object there is parallax disparity in the images that the left and right eye sees due to the interocular distance between the eyes. The human brain, as well as that of some other creatures, fuses the two views together forming a composite stereo image, hence the real world is viewed in stereo. When the person, the scene, or both together move, changes due to parallax, perspective, diffuse light and reflections in the scene occur because of the relative change of viewpoint. It is possible to look around objects revealing more of the object and more of the scene behind the objects. These changing views are in stereo and during movement continuous seamless stereo views are presented to the human imaging system. Autostereoscopic displays that try to emulate real world natural viewing require numerous simultaneous samples of the plenoptic function for a single image. This requires an increase of information about the scene when compared to that of a standard photograph that is a monoscopic image of a single sample, and in the field of computer graphics natural viewing emulations are computationally very demanding.

If these samples, simultaneously present on a display medium, can direct the light in correct but different directions to provide left and right eye changing stereo views, then the task of imaging a parameterised plenoptic function will be achieved. The extent of the parameterisation is logically reflected in the seamless look-around quality of the displayed image.

An optical medium that solves this problem of directionality is the microlens array. The arrays can contain hundreds of semi-spherical, hexagonal, or semi-cylindrical lenslets. These are placed over the 2D intensity distribution and direct the light in various directions. Usually, the focal plane of each lenslet is on the back surface of the lens array media and it is here that the intensity distribution, inherently containing directional information, is located. Registering the lens array correctly on top of the 2D intensity distribution enables left and right eye disparate zones (windows) to be refracted to the viewers. Changes that the moving viewer sees in the image is dependent upon the method used to position the intensities on the 2D distribution. These methods can broadly be categorized as, integral and multiview imaging. For the latter there is a profundity of literature and while it has many similarities to integral imaging major differences exist and a short review and a comparative explanation of multiview are necessary in order to define and clarify integral imaging.

1.4 Multiview

Compositing of 2D views, whether spatial or time-multiplexed, is the technique of multiview imaging and the depth of the display is implicitly encoded as positional disparity between displaced 2D views. It has come under close scrutiny in respect to the psychological and physiological effects on viewers especially due to a mismatch of convergence (change of eye aim when an object's distance changes) and accommodation (change of eye focus when an object's distance changes) [6]. When the brain adjusts to the display, after prolonged stereo viewing, dizziness and disorientation is often the outcome.

Generally, multiview systems produce multiple viewing zones and the viewer is positioned so that each eye falls within a stereo pair of zones. To maintain a correct

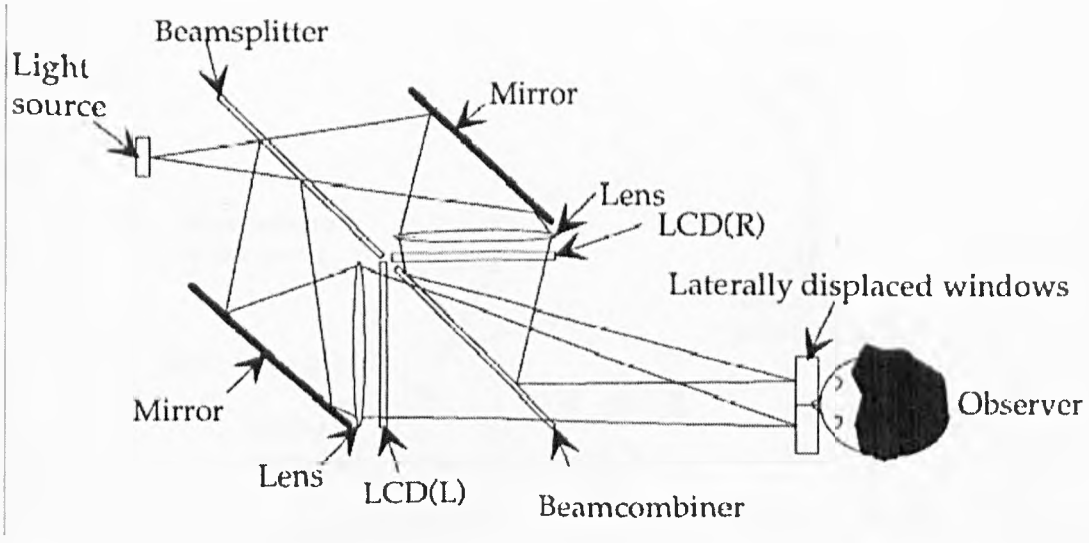

Figure 1.1: Optical schematic of Sharps 'Twin-LCD' full-resolution autosteroscopic display

viewing condition, as the viewer moves laterally adjacent zones are available. These multiple simultaneous views have a high bandwidth requirement as they use a large number of views to maintain a 3D image over a wide angular range. A system using nine zones from nine video projectors [7] and a sixteen zone time-multiplexed system [8] have been reported. Systems in which the image bandwidth requirement is more relaxed use dynamic steerable zones in which a small number of zones are moved in correspondence with a measurement of observer position [9]. In these systems only two views per viewer are required to be displayed at any one time. All of these multiview systems, though, do not produce image look-around and 'flipping' is a well known term within the multiview community to describe what happens when the viewer moves into adjacent stereo viewing zones. The relatively large distances of the projectors from one another result in flipping due to gaps within the plenoptic field and consequently all round viewing is not continuous. It has been reported that lateral look-around requires more than 60 views across the interocular distance [10]. Sharp laboratories have tried to circumvent the problem of image look-around and bandwidth limitation by introducing their 'Twin-LCD' display [11][12][13] (Figure 1.1). Each panel displays one of the stereo pair images and observer tracking is implemented by laterally moving the light source, which, in turn, moves the image zones. Importantly, each eye sees an image with full panel resolution. Additionally, by using two light sources two viewers can be accommodated. Two video laser disk

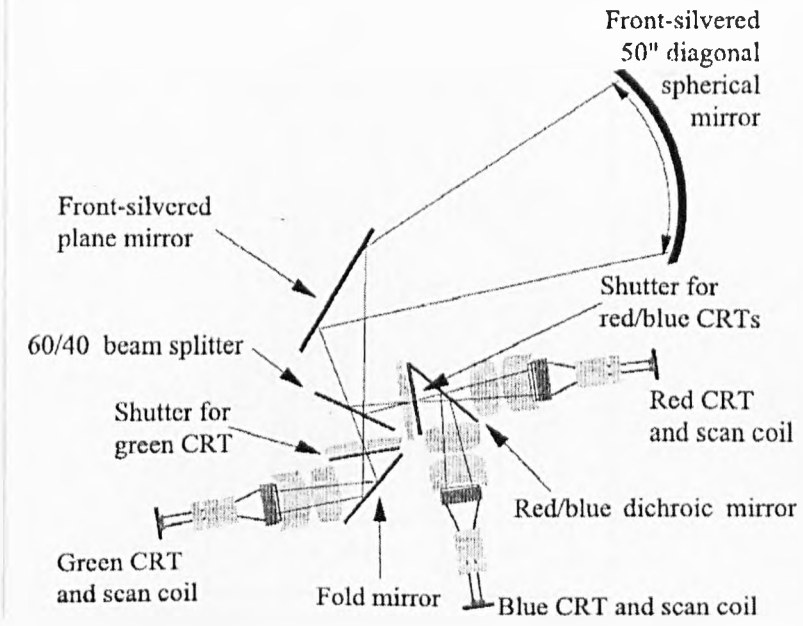

Figure 1.2: Optical schematic of the Cambridge 50" 15-view multiview display

players are used to generate two video channels of pre-recorded scenes captured by a pair of professional cameras.

Obviously, the complexities of two panel displays impose a limitation on the range of accessible consumer applications and the one or two viewers must be at the optimum distance from the display. Sharp later focussed their research on a single panel parallax barrier LCD display that can convert from 2D mode to autostereoscopic 3D mode but without the look-around facility [14].

When using microlens arrays LCD flat panels are usually used. However, there are multiview type displays that have been built for CRT's that do not use microlens arrays. The Cambridge monochrome display [15], for example, works by lifting the image off the CRT with a compound image transfer lens containing a ferroelectric liquid crystal shutter element. A Fresnel lens is placed at the plane of the transferred image. Using time-multiplexing, displaying displaced views in turn and only allowing strips of each view to be used by making segments of the shutter transparent in synchronisation with the views "gives a smooth stereoscopic effect". Their colour display is much more complex requiring three CRTs and beam splitters (Figure 1.2).

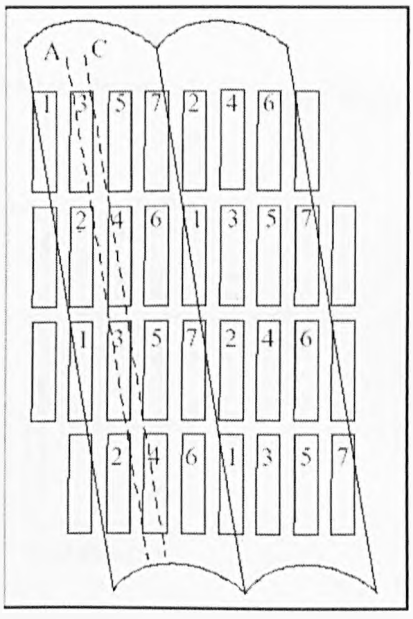

Figure 1.3: Philips' slanted lenticular 3D-LCD

For a single optical system only one viewer at a time, in the correct position, can use the device. The system displays 15 views to make a composite image at 640x480 resolution and 30Hz interlaced refresh rate and the viewing zone (aperture) is 330mm hence giving 22mm separation between views. It can provide 30s of animation using 15 pre-rendered 2D image sequences.

A benefit of *not* using LCDs is that moiré effects are not encountered. Moiré patterning is a major problem when viewing the display and results in dark vertical bands appearing in the image due to the different spatial frequencies of the lenslet and pixel interfaces. Even if an exact integer number of pixels are present behind each lenslet then colour moiré is still present in which the spatial frequencies concerned are those of lenslets and the RGB components of each pixel (i.e. sub-pixels). Another way of expressing this is that the lenses magnify the pixel interfaces out into the viewing zones. Philips [16] managed to avoid the problem by slanting the lenslets at a small angle to the vertical axis of the display. This has the effect of 'dissolving' the moiré bands. Figure 1.3 diagrammatically shows the implications of this for a 7-view system [17]. The numbers represent the view number that the individual RGB sub-pixel belongs to. Lines A and C show the selected sub-pixels for views 3 and 4.

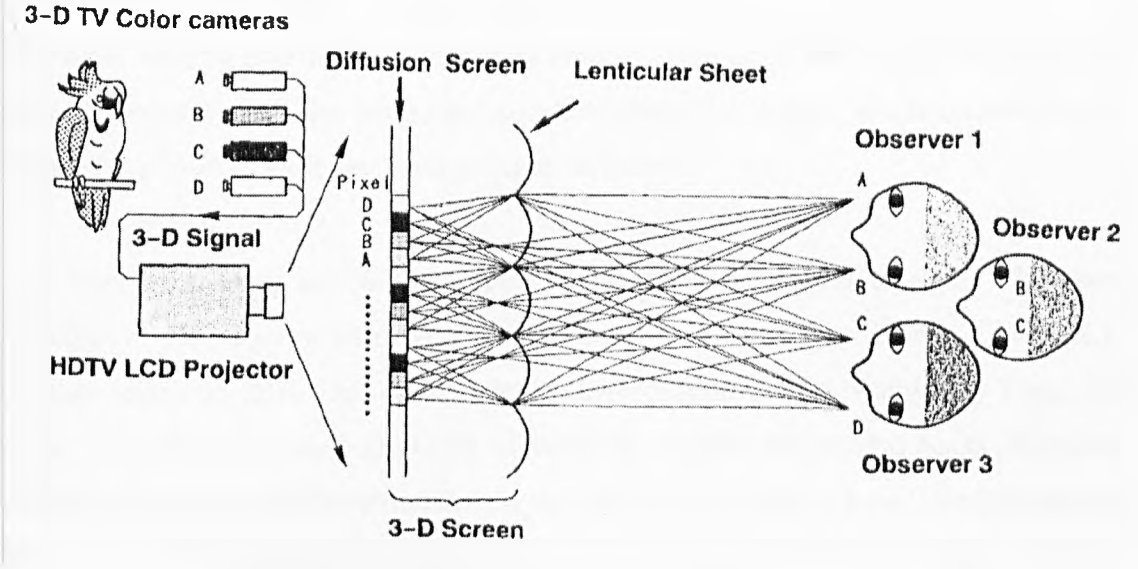

Figure 1.4: Optical schematic of the NHK 50" 4-view multiview display

---

Normally the horizontal resolution decreases in direct proportion to the number of views and the vertical resolution remains the same, but the Philips 7-view system provides an overall resolution reduction both horizontally and vertically by factors nearing $\sqrt{7}$. They are presently optimising their multiview rendering capabilities.

NHK Japan produced a 50" multiview full-colour projection display [18], using an LCD projector, eight years prior to the Cambridge CRT based 50" display. In their display (Figure 1.4), images from four NTSC colour TV cameras are electronically multiplexed, enlarged and rear-projected by a HDTV LCD video projector onto a lenticular sheet [19] fitted with a diffusion screen. The LCD projector incorporates three 5.5 inch a-Si LCD panels [20][21] contained as a single compact and lightweight projection unit giving a combined resolution of 4.5 million pixels. Obviously, the quality of the displayed video, which depends on the number of views and the actual achievable display resolution of the display device, is limited, and a report [22] from an Open House viewing of the display read:

"...interesting but not spectacular. Since then, the number of cameras has been doubled to eight. The camera lenses are essentially touching so there is not much room for further improvement here without major optical changes. The resulting

images are much improved in their 3D quality, while remaining bright and highly viewable. Seating position is constrained both as distance from front of the screen and lateral movement but the additional cameras made the images more interesting and reduced the need to shift my head as much as before."

It is clear from these selected examples of autostereoscopic displays that some have benefits in one area whilst others have benefits in another. Sharps 'Twin-LCD' display seems to fulfil look-around and full-resolution qualities and they claim that there is no reported visual strain by viewing the display for several hours. However, as with all stereo-based systems, the display is only available to a very limited number of viewers at any one time. However, observer tracking over a wide angle is available. The system is seen as a test-bed for future 3D research and is an interesting concept but would mainly be reserved for experts and researchers and not be commercially viable for general public use. It is a similar situation for the bulky Cambridge single-viewer display. The Philips display, though it solves the problems of moiré patterning for systems using lenticular sheets and LCDs, and balances out unequal resolution parameters horizontally and vertically (though there has been no reported problems with the unbalance), only multiplexes 7 views, and hence either "flipping" or a limited attainable depth resolution results.

What, then, are the optimum requirements for a 3D system in terms of acceptable 3D viewing and commercial availability to the general public. Television and PCs are the layman's main equipment for viewing images in the home. 3DTV, 3D user interfaces, 3D web browsing and 3D worlds on the PC are vast commercial targets for a viable 3D system and as both TV and PC come together with the dawning of digital TV, transmission requirements can be simultaneously fulfilled for both. The TV, at least, is generally used as a group viewing medium and a requirement for 3DTV must be that it is multi-viewer capable. Autostereoscopic imaging produces displays that do not require special headgear or glasses and the move by most researchers is in this direction. The real world is seen without any "flipping", with all-round viewing, and without visual strain; these must be other requirements. TV viewing does not constrain viewers to watch from a pre-ordained viewing distance and consequently, flexibility in viewing position is necessary. In fact, the optimum requirements are those that are compatible with present day monoscopic television viewing. Anything

less would not be acceptable to the general public except as a passing fad. Researchers at DMU recognized the potential of integral imaging and it is slowly gaining momentum as a possible major contender for 3DTV. It has the potential to fulfil all of these optimum requirements but has been little researched when compared to stereo/multiview systems.

1.5 Integral imaging

Lippmann [23] first submitted the optical principles of integral imaging in 1908 followed by further work by Ives [24]. The basic principle is the capture of the scene by an array of vertically and horizontally aligned microlenses. Each microlens captures its sub-image at its focal plane, which is the same plane for all the microlenses in the array. Each microlens has its own viewpoint of the scene and these viewpoints together generate an intensity distribution on a single capture medium. When a microlens array, with the same geometries as that used to capture the scene, is correctly registered on the intensity distribution the scene is volumetrically replayed into 3D-space as a pseudoscopic (axially inverted) copy of the original scene. A further capture and replay of the copy generates an orthoscopic (natural) scene. The classical approach is a two-stage process and this limited the usefulness of the technique. The capture mechanism, however, uses a single aperture plane (the lens array) and as such does not require multiple cameras and projectors. The technique also generates omni-directional parallax and look-around is completely seamless due to the microlenses abutting each other. From a plenoptic function point of view, the extent of the samples of the scene captured, and hence the quality of the replayed integral image, is dependent upon the size and number of microlenses and the resolution of the capture/replay medium. In comparison to multiview the number of samples of the scene has increased many times, there are no gaps between samples and hence no "flipping" artefacts.

Lippmann's original technique captures objects that are relatively distant from the lens array (remote imaging) and on replay all of the object or scene is in front of the array. Due to a limited depth resolution capability of the technique (see sections 2.2.1, 4.3.2 and 7.4) a better set-up is to capture and generate the scene as it straddles the

array (close imaging). This provides an integral image that replays in front of and behind the lens array and it is much easier for the scene to be re-integrated.

Uni-directional, horizontal parallax integral images can be made using lenticular arrays and, as viewers tend to move their heads from side-to-side more than up-and-down, it becomes a version of integral imaging that is reasonably satisfactory. Omni-directional or uni-directional parallax integral imaging has multi-viewer capability and generates a main front-viewing lobe and smaller periphery lobes.

An important beneficial phenomenon of integral imaging is that accommodation and convergence are the same as when viewing the real world, and viewing orthoscopic integral images therefore present none of the uncomfortable viewing traits associated with stereo/multiview. The explanation for this effect is that many adjacent microlenses image each object point and on replay the ray bundles produced by these microlenses intersect at each corresponding image location, relevant to the viewers position, in 3D image space. The eyes then focus and converge to these positions (Figure 1.5). As the viewer moves laterally new groups of bundles from the adjacent pixels intersect providing a look-around facility. It is possible to retrieve a continuum of 2D views by software if the intensity distribution is digitised and then low-pass filtered to avoid aliasing artefacts [25]. A disparity analysis can then be performed to achieve reliable depth estimates. This leads to the ability to selectively extract objects and effect synthesis with other integral images in real time.

When decoding lens arrays are placed over integral distributions the interfaces of the microlenses can be seen when viewed at short distances. However, the smaller the microlenses the more the interfaces disappear from view, and when very small good quality microlenses can be manufactured the problem will no longer exist. Recently intensive work has come close to resolving the technical difficulties associated with their manufacture. This work includes producing arrays using photoresist carried out at DMU [26] and another technique developed within the LAIRD project [27] for producing high quality hexagonal or square based lenslets.

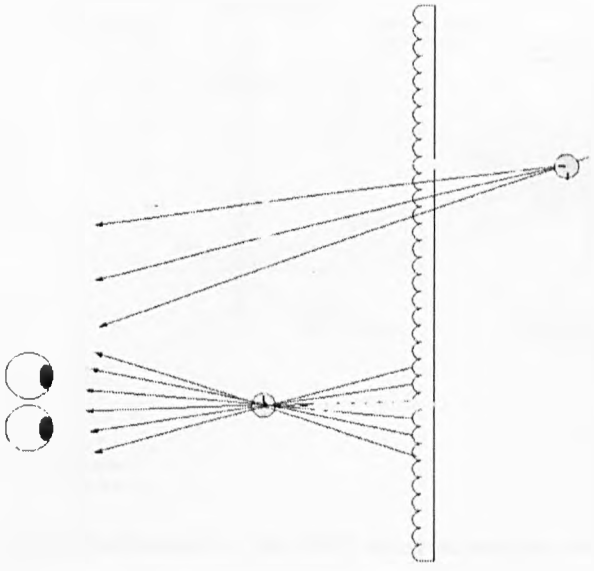

Figure 1.5: Ray bundle intersections providing accommodation and convergence match

In 1988, Davies and McCormick [28] developed an innovative optical approach that not only overcomes the problem of the two-stage process but also enables close imaging. This system allows it to become a single-stage process by the use of an optical transmission inversion screen (Figure 1.6) that inverts the axial spatial sense of an object, projecting a pseudoscopic image for encoding. The double integral screen acts as a direction-selective field lens transmitting the rays at equal and opposite angles. The pitch of the microlenses governs the lateral resolution and the large apertures of the macrolens arrays ensure the retention of the depth resolution. Aberrations induced by the input macrolens array are, to a large extent, cancelled by the output macrolens array. A microlens array placed within the reformed pseudoscopic scene samples this 3D space and the lenticular encoded spatial distribution (LeSD) can be captured electronically or on film.

The completed MkII camera (Appendix A) can be seen in Figure 1.7. Each of the two large area macrolens arrays (Figure 1.7 front view) simulates a large single aperture in conjunction with the microlens transmission screen. The individual injection moulded macrolenses have a hexagonal base and were produced to a high level of precision;

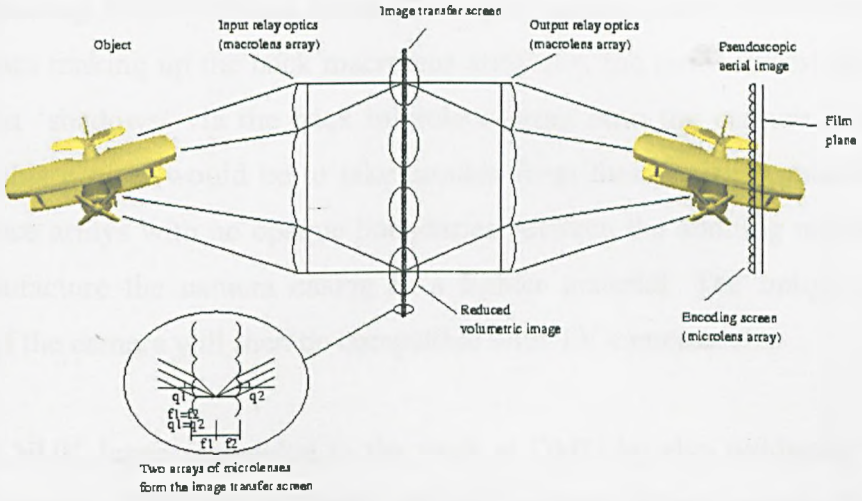

Figure 1.6: Optical schematic of the DMU integral imaging camera[1]

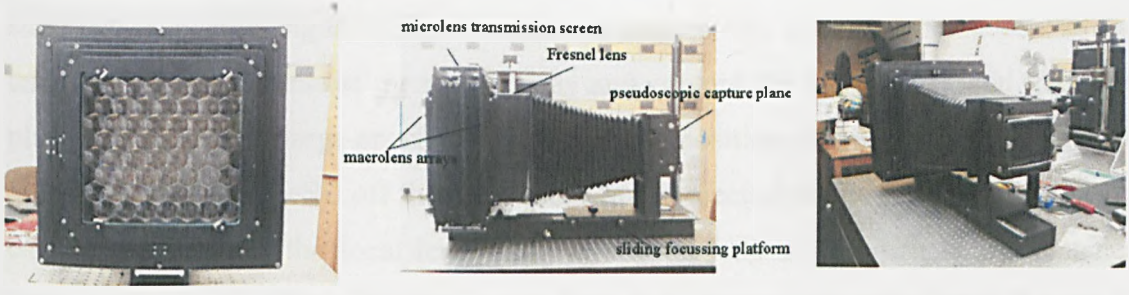

Figure 1.7: DMU integral imaging camera. Left: front view. Middle: side view. Right: back view.

this being a necessity to form an accurate full-fill array. Laser alignment techniques were developed to achieve a final image integration capability with a minimum accuracy corresponding to 20 seconds of arc. An assembly rig was designed to enable tilt and yaw adjustments to be made and thereby achieve a higher level of planarity between the macrolens arrays [27]. The Fresnel lens acts as a depth control lens and can set the image size for electronic capture, it can also easily slide out of the camera body to allow photographs with a 1:1 ratio to be produced. The parameters for the optics of the camera were evaluated using optical matrix equations as described in Appendix A p157-159.

The integral images are captured at the pseudoscopic capture plane. The plane can be moved on a sliding focussing platform thereby selecting the portion of the spatial

[1] Figure courtesy of Matt Forman, DMU

scene appearing in front of and behind the replay screen plane. The interfaces of the macrolenses making up the back macrolens array (i.e. the array closest to the capture plane) cast 'shadows' via the back microlens array onto the capture plane. Further work on this project would be to take moulds from these existing macrolens arrays and produce arrays with no opaque boundaries between the abutting macrolenses and also manufacture the camera casing in a lighter material. The unique design and concept of the camera will then be compatible with TV cameras.

Recently, NHK Japan responded to the work at DMU by also producing an integral imaging camera. They use a 'direct pickup' method (direct capture method) that captures each image field directly via a television camera (a lens that projects images to a pixellated charge coupled device – CCD) and the LeSD is transmitted as a standard television signal [29]. In order to reduce the necessary distance of the television camera from the microlens array and capture the LeSD as it is at the focal plane of the array a large-aperture convex lens is positioned immediately behind the array. This lifts the LeSD off the focal plane and projects it to the camera. This works correctly as long as the focal length of the convex lens is the same as the distance between the convex lens and the camera lens. It is a standard method that is used in some multiview techniques, for example, the Cambridge display (Figure 1.2) lifts the images off CRTs using compound image transfer lenses. A problem they seem to come across, though, with this method is that the image fields tend to overlap generating interference and the images remain pseudoscopic. To overcome the difficulties of the latter NHK devised a plan to use two microlens arrays with matching microlens optical centres, much like DMUs image transfer screen. This has the effect of inverting the image fields to produce a final orthoscopic integral image at their display. For some reason overlapping image fields remained a problem for them and they decided that optical barriers were necessary between the individual microlenses. To this end they decided to employ the use of two horizontally arranged gradient-index lens arrays (multimode fibres or GRIN rods) [30]. Using specific lengths of gradient-index lenses erect orthoscopic images can be displayed (Figure 1.8) and by having an internal reflection angle of 1.5π it was found that no overlapping of image fields occurs. The resulting integral images require higher display resolutions and the gradient-index lens array provides a limited depth of focus.

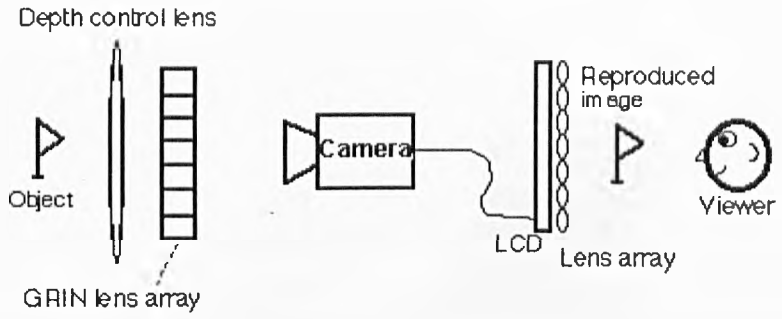

Figure 1.8: Optical schematic of the NHK gradient-index lens array integral imaging method

Seoul National University, Korea are also beginning to look into the possibilities of using integral imaging for 3DTV and are producing analytical papers in coordination with NHK and with the University of Connecticut [31][32][33].

1.6 Computer generation of integral images

S.Min et al at Seoul National University reported on computer generated images [34] and present basic experimental results for a single object produced on a 65mm x 65mm display area. They produce images that are displayed either entirely in front of the array or entirely behind the array and do this by calculating (mapping) each pixel hit relevant to each 3D object point at a time. Gradually the image fields are filled when all the points are mapped. Nothing has appeared from them on this matter for nearly three years. Another paper by T.Naemura et al [80] from the University of Tokyo with the initially misleading title of "3-D computer graphics based on integral photography" explains the synthesizing of arbitrary 2D views from the intensity distribution captured by the NHK system. This is a technique that has been performed and reported many years previously by Adelson et al [25], and is a standard technique that DMU use to verify the quality of photographic integral images and study the nature of continuous parallax in pixelated integral images [35][36]. Igarashi et al [37] and Chutjians et al [38] made previous attempts at the computer generation of integral images but these were of simple wire-frames and the software models used to simulate the lens sheets were very basic approximations. Additionally, the displays had a limited resolution.

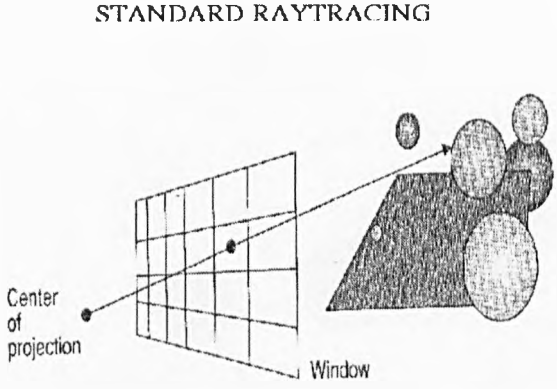

Figure 1.9: Standard ray tracing

A significant amount of research has been reported, by DMU, on the computer generation of integral images [39][40][41][42]. This has been based on using ray tracing, a well-known method used in standard computer graphics to produce photorealistic images.

1.6.1 Standard ray tracing

Standard ray tracing is an expensive method due to the cost in computing time and it also suffers from aliasing problems. In standard ray tracing the centre of projection is positioned behind a virtual pixelated plane and from this centre at least one ray per pixel is cast out through all the pixels in turn into 3D object space (Figure 1.9). Each ray searches for intersections of objects along the path and algebraic equations for the ray and object are computed. These equations are dependent upon the nature of the object e.g. intersection equations of line-sphere, line-convex polyhedron, line-box etc. In scenes of moderate complexity approximately 95% of processing time is spent in intersection calculations [43]. The ray calculations are independent from each other, however, and this allows for an easy implementation of parallel processing hardware. Once an object has been intersected and the colour contribution assessed by direct and ambient light, reflection and refraction direction calculations are made. This generates two rays that can now search for other weighted colour contributions for the pixel involved and if these find intersections then they again can split into two directions and search again (Figure 1.10). This can progress for as many levels as required but

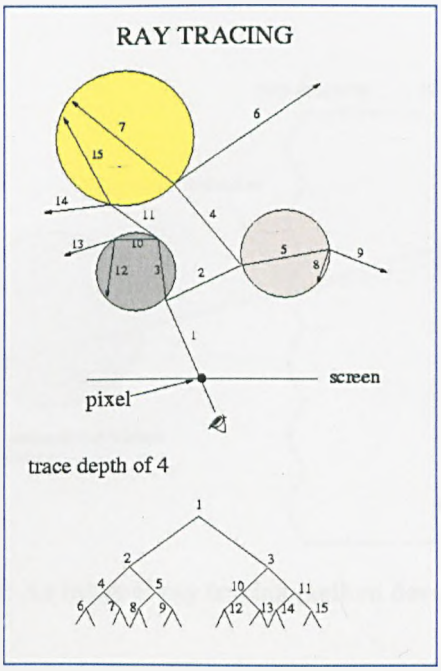

Figure 1.10: Trace depths in the ray tracing technique

later intersections are weighted less [44]. An immediate benefit of ray tracing is that inter-object reflections are a natural outcome of the technique.

1.6.2 Integral ray tracing

The basic integral ray tracing technique developed at DMU requires that a pinhole model of a lens array covers the pixels that are at the focal plane of the lens array. Each pixel belongs to an image field of a particular lenslet and rays are cast from the pixel out through the pinhole of the relevant lenslet (Figure 1.11). However if the plane of pinholes is moved to the plane of the centres of curvature of the lenslets then no refractive elements need to be considered as refraction does not occur for rays travelling through this centre in a real lens. To perform 'close imaging' it is necessary to cast rays backward behind the array to search for intersections. The forward projecting ray searches for *last* intersections as these positions are the parts of the surface closer to the viewer, also correct occlusions result from this procedure.

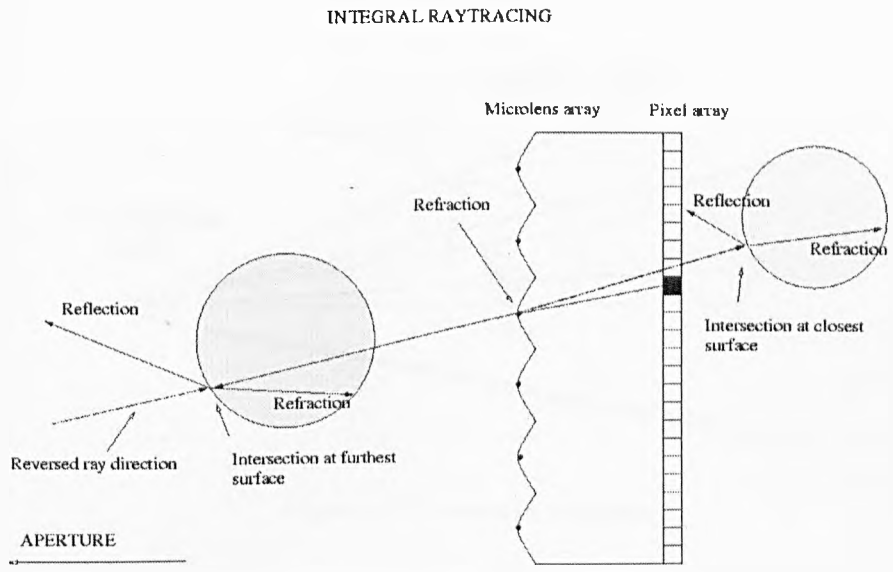

Figure 1.11: An integral ray tracing method developed by DMU

1.6.3 Standard interpolative shading

The standard rendering technique that is used for fast image processing calculates pixel intensities for projections of the corners of polygons (usually triangles) and by a process of bilinear interpolation estimates the pixel values across a horizontal span. The percentage of estimation depends on the size of the triangle.

Figure 1.12 shows the basic methodology of the technique in which a single projection point creates a viewing frustum within which the contents of the scene are rendered. The object exists as Cartesian coordinates in the object file depicting a mesh of triangles that form the objects within the scene. The corners of the triangles are projected to a pixelated image plane and the pixel colours are calculated and interpolated revealing a rendered perspective view of the scene.

The usual method of calculating the pixel intensities is one that has found wide acceptance in the computer graphics community, namely the Phong illumination model or the Phong reflection model that contains a combination of diffuse, specular and ambient components:

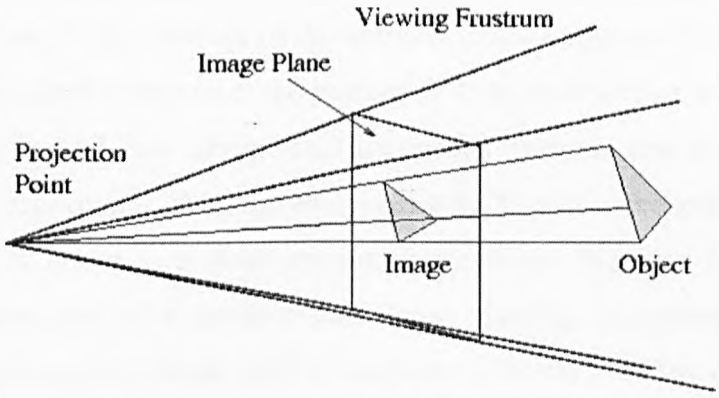

Figure 1.12: Standard projection

$$I_r = I_a k_{ar} + I_i\left(k_{dr}(L \cdot N) + k_s(N \cdot H)^n\right)$$
$$I_g = I_a k_{ag} + I_i\left(k_{dg}(L \cdot N) + k_s(N \cdot H)^n\right)$$
$$I_b = I_a k_{ab} + I_i\left(k_{db}(L \cdot N) + k_s(N \cdot H)^n\right)$$

where:

$I_a k_{a(r,g,b)}$ is the ambient component and is an approximation to the global illumination
I_i is the intensity of a point light source
$k_{d(r,g,b)}$ is a wavelength-dependent empirical reflection coefficient
L is the light direction vector
N is the normal to the surface and reveals the orientation of the triangle in 3D object space
k_s is the specular contribution
n is an index that simulates surface roughness
H is the unit normal to a hypothetical surface that is oriented in a direction halfway between the light direction vector L and the viewing vector V i.e. $H = (L+V)/2$. [45]

There are two interpolative shading techniques that are commonly used today; these are the Gouraud interpolation method [46] and the Phong interpolation method [47]. There have been other more simplified techniques during the development of the first shading schemes one of these being what is called 'flat shading' (constant shading) [48]. In flat shading a single intensity is calculated for each triangle and all the pixels within it. Another, even more simplified technique, has its initial intensity values

based on the distance from the viewpoint followed by interpolation [49]. The Gouraud and Phong shading methods require that the normal N in the illumination equations is the vertex normal i.e. the average of the normals of the polygons that share the vertex. The Gouraud method interpolates the intensities thus calculated at the triangle corners and the Phong model first interpolates the vertex normals and then calculates the intensities for each pixel. This allows specular highlights to be picked up for inter-triangular pixels but it is a more expensive technique than the Gouraud method. Gouraud shading, and to a lesser extent Phong shading, is incorporated into most hardware graphics cards along with a hardware Z-buffer (hidden surface removal). These hardware speedups allow real-time rendering and this is something that will probably evade ray tracing methods for a long time.

1.6.4 Integral interpolative shading

A classical integral imaging system based on the ray tracing method would effectively require one view to be produced per lens. This produces a computer intensive problem, as the solution is dependent upon an enormous amount of high precision floating point arithmetic. To create a flexible real-time system, an alternative solution to this problem is sought using a fast interpolative shading methodology. The reasons to attempt to accomplish this are many, ranging from real-time medical usage to interactive integral computer games. In fact, for all the same reasons that real-time computer graphics has evolved over the years. This would provide volumetric video, improving the present 2D representation of 3D by a further dimension. The main objective of the research presented in this thesis is thus the production of computer generated integral images using an approach similar to that of the standard forward geometric projection technique that uses interpolative shading methods.

1.7 Forward geometric integral projection development criteria

Outline development strategy aims are proposed that will move the research through stages where each stage will bring more information to bear on the problems associated with reaching the main objective. Essentially, these aims will follow two major paths, a pinhole model approach and a full aperture model approach. The former is expected to be the way towards the development of real-time computer

generated integral images. The latter will compliment research ongoing in the development of a real integral camera and in its own right be a useful tool for producing computer generated integral images. The strategy is listed below:

- Develop model structures as a base for the production of integral images using forward geometric projection techniques.
- Initially, develop computer generated mesh integral images for both pinhole and fully apertured lensed arrays and establish volumetric qualities with respect to the effects of block pixelation on the nature of the reconstituted image.
- Investigate existing fast shading algorithms such as those developed by Gouraud and Phong and extend to operate in 3D space to allow the creation of solidly filled, illuminated optical models.
- Develop rendering software to emulate a real integral camera by modelling fully apertured lensed arrays and investigate the effects of spherical aberration thus produced.
- Develop rendering software that will generate integral images by modelling the lenslets in a lens array as pinholes.
- Investigate requirements of the computer generated integral images displayed on LCDs and produce integral off-line video displays. Use high-resolution integral images in projection experiments.
- Ascertain the requirements for a real-time integral imaging system.

The outcome of these strategies are found in the following chapters:

In Chapter 2 the optical capture and replay mechanism of microlenses is analysed in order to enable the development of strategies to generate integral images by interpolative shading techniques. A thorough examination of the action of a microlens array is performed followed by the development of suitable methods of implementing integral images from this analysis. This includes general equations to prove that the ideas subsequently used are structurally sound.

Chapters 3 and 4 explain two *non-shaded* pre-emptive attempts made to ascertain the validity of a different approach to that of ray tracing. The first, in Chapter 3, is a pinhole model that aims to produce simple mesh (or spot) integral images. The second, in Chapter 4, is the modelling of finite-sized lenslet apertures within a lens array, a method that again attempts to produce integral mesh images but brings the problems of spherical aberrations to bear as in real optics. Both of these aims are to provide an insight into the problems associated with full integral rendering techniques. The aim is to produce volumetric *mesh* images with all-round viewing and understand and overcome obstacles encountered that could possibly aid in the development of *rendered* integral images.

The rendering techniques, based on the analysis of Chapter 2, are described in Chapters 5 and 6 and are of a finite-sized aperture lens array model and a pinhole model inclusively. The aim of both of these models being to provide fully rendered and shaded volumetric images with all-round viewing. An added bonus at this stage would be to produce off-line integral animations. Real-time implementation is a final goal, but within the confines of this research in terms of time and money, the extra processing power and hardware requirements do not make this a viable consideration.

Chapter 7 provides a variety of examples of image production and display.

Chapter 8 outlines methods that are required to produce real-time integral imaging and experiments to show proof-of-principle of this analysis are included.

Finally, Chapter 9 presents the concluding overall view of the research and to what extent the aims and objectives were achieved.

2

Forward Geometric Projection Capture Methods

2.1 Introduction

This thesis is concerned with the computer generation of integral images and especially with the theory and practical development of a forward rendering projection technique. Standard raytracing is known to be a very slow process but has the benefit of photo-realistic output images. The alternative is the forward projection method whereby most of the pixel intensities are estimated by bi-linear interpolation. This method is used in standard computer graphics for real-time generation and the main purpose of the research explained within this thesis is to provide the methodological and practical requirements for the generation of integral images in real-time. The spatial information collected at the image plane via the lens array to produce a LeSD is much more than for a standard 2D image and for this reason it is necessary to evolve fast methods yet still enable the image to retain its integral nature. This chapter shows the arguments leading to the development of two possible options and two methods derived from these options.

2.2 Ray mesh pattern produced on playback

2.2.1 Simplifications of mesh representation and beam spread

Analysing the way in which light is replayed through a lens array from an arbitrary focal plane gives clues as to how a suitable capture method might be derived (Figure 2.1).

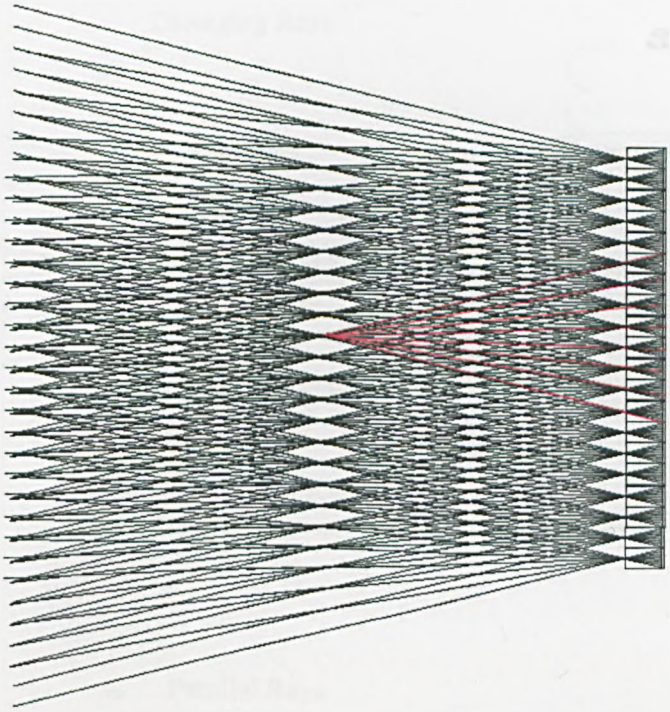

Figure 2.1: Pixels replaying through a lens array

The diagrammatic representation of light rays replaying through a lens array, shown in Fig 2.1, is based on a very simplified reverse pinhole approach where each lenslet pitch is an exact integer multiple of the pixel pitch. In practice it would be both difficult and expensive to produce on a real system. Each pixel fills the whole relevant lenslet aperture and is refracted and magnified out into viewer space. The *ray bundles* individually spread out as they move further into viewer space reforming the original point as a large diamond shaped voxel (see Figure 2.2). The voxel size represents one of the limits imposed on the depth resolution capabilities of an integral imaging display system. If quality aspheric (ideal) lenses could be manufactured then the diamond would be restricted in size as the generating ray bundles beam spread would be reduced and the depth resolution thereby improved (see Figure 2.3). It can be seen (see figure 2.4) that the angle of divergence is dependent upon the pixel size and the distance between the pixel and the centre of lens curvature. Smaller pixel sizes and smaller pitch lenslets would therefore also aid in resolving the problem of depth resolution by generating smaller replayed points. The light rays from the pixels also replay through adjacent lenslets and this allows side lobes to be formed either side of the main lobe; this is ignored in the light ray representation.

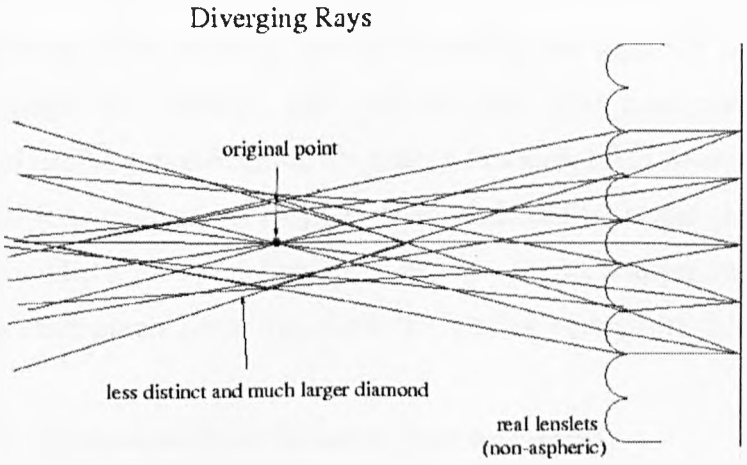

Figure 2.2: Beam spread with normal lens arrays

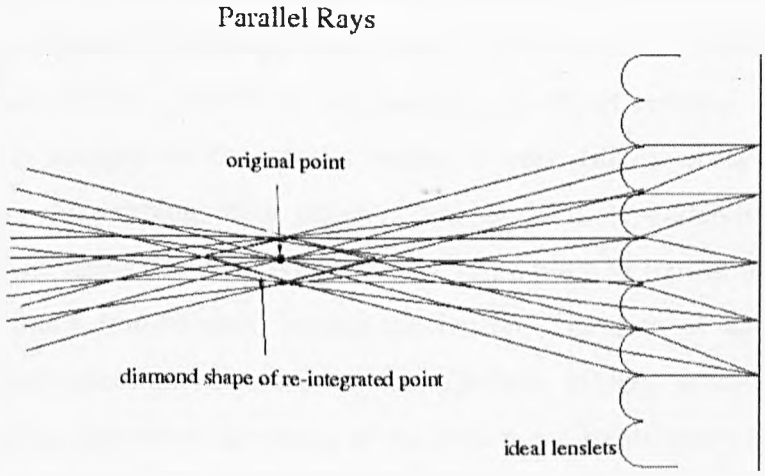

Figure 2.3: Beam spread reduction using aspheric lens array

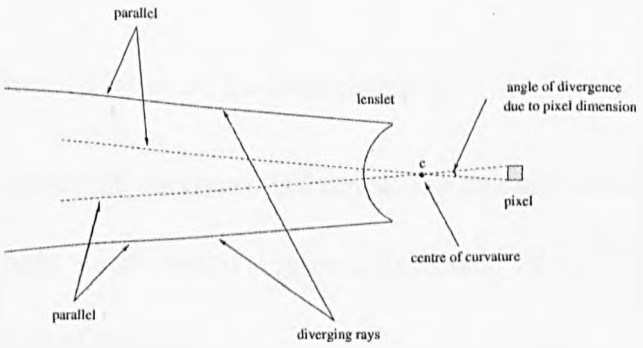

Figure 2.4: Ray divergence due to pixel dimension

Diagrammatically to represent reality would be too complex and by only allowing the centre of the beam to be included, that is by passing the light ray from the centre of the pixel through the pinhole, and without side lobe configurations, gives an acceptable and useful simplification. To enable this simplified interpretation to work, for a particular lens system, the pinhole location should be at the centre of curvature of the lenslets where no refraction takes place. As the viewer moves around the display consecutive pixels come into view, the lenslets acting like directional pixels.

2.2.2 Aperture planes and their distance from lens array

To analyse more closely the possible geometric configurations for perspective capture of a LeSD it is first necessary to look at the image fields behind each lenslet and particularly behind the central lenslet. Depending upon the lens array parameters of lenslet pitch, surface of curvature and focal length, there is a fixed boundary for aperture planes to exist, parallel to the lens array in object space (Figure 2.5). When the aperture is imaged by the central lenslet it must fill the image field correctly without either overlapping other adjacent image fields or under-filling the central image field. The image field is taken to be the same pitch as lenslet pitch and situated at the focal plane immediately behind the lenslet. The size of aperture used (A), therefore, determines the distance of the aperture to lens array (d), and when calculated for the pinhole at the vertex of the lens, it can be shown to be:

$$d = \frac{A}{2\tan(\arcsin(n_2 \sin(\arctan(\frac{P}{2f}))))}$$

where n_2 is the refractive index of the lens material.

Conversely, if the centre of curvature is known, the calculation can be simplified by positioning the pinhole at this centre (Figure 2.6), hence: $d = \frac{A(f-r)}{P} - r$ where r is the radius of curvature of the lens.

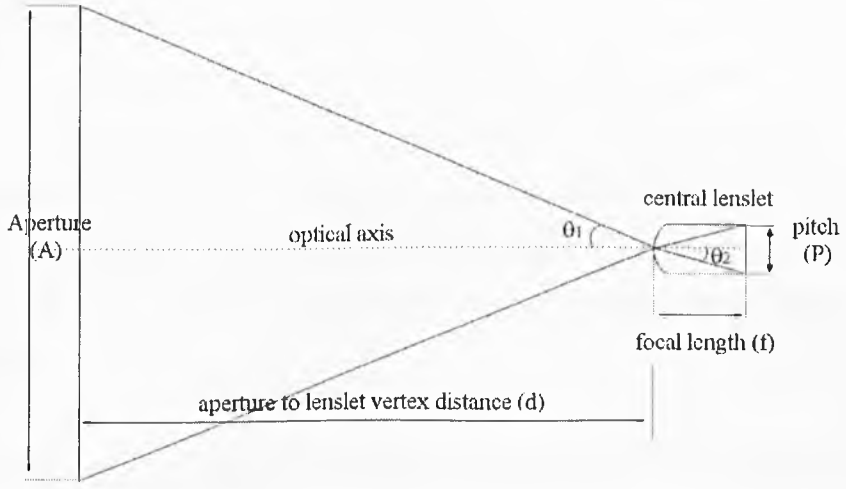

Figure 2.5: Aperture to lens array distance calculated with refraction

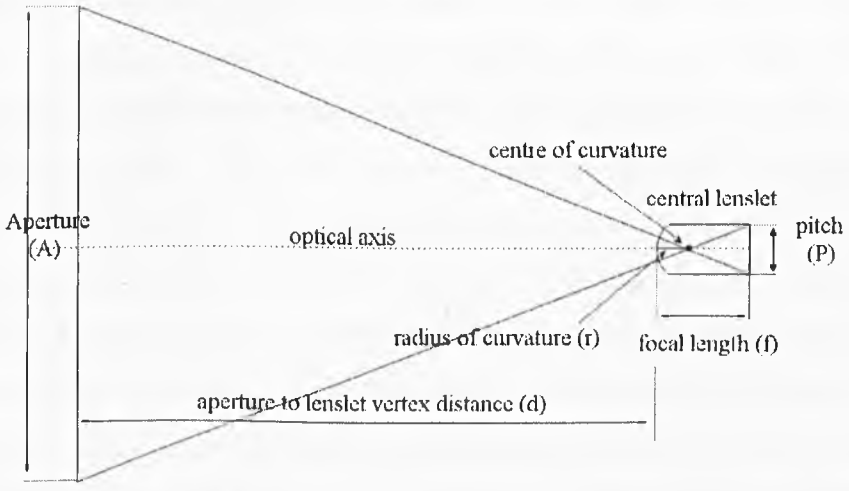

Figure 2.6: Aperture to lens array distance calculated without refraction

2.2.3 Problems of using the light-ray mesh for a capture technique

It is convenient and accurate to use and modify this model of replayed light rays (Figure 2.1) to represent the action of Lippmann photography where the scene is captured by a lens array and during display is replayed back to its original spatial and volumetric position. Although direct capture and replay produces a pseudoscopic integral image, when generating integral images by computer graphics there is no difficulty in arranging orthoscopic replay.

Analysing the diagrammatic light ray structure it can be seen that all rays from similarly positioned pixels relative to each lenslet are parallel. This immediately assumes an orthogonal mode and is reminiscent of orthogonal ray tracing or orthogonal forward projection. These are techniques in standard computer graphics that do not require perspective imaging e.g. architects drawings. Objects with the same sizes will be the same size in the final image irrespective of the depth difference between them.

Again from Figure 2.1, it can be seen that at particular depths, in object/image space, rays come together at major and minor points in the same plane. The major planes form at integer multiples of d and the number of rays which meet at each of these locations is the same as the number of pixels behind each lenslet. Each of these points can be visualised as projection points or pinholes through which a previously captured conic volume of a scene – but with the scene now taken away - is being projected. Each projection is forward facing and has the same fixed field of projection as the other projections in the same plane. The angle of the field of projection is the angle swept out from the point to the extremes of a number of lenslets. The number of lenslets involved is the product of the number of pixels behind each lenslet (n) and the integer number of distances of d used for the projection plane. This is not an exact statement as there is a slight variance from this depending upon whether there are an odd or even number of pixels per lenslet. For example, for the plane at distance d, with an even number of pixels per lens, the number of lenslets is n, and at $2d$ is $n-1$, at $3d$ n, and at $4d$ $n-1$. This interchanging value reflects the position of the projection points relative to the lenslets, either centrally located or in line with the interface between lenslets. Another related fact is that projection points at the first major plane send their rays out to adjacent lenslets while those at the second major plane intersect every *other* lenslet and this increasing gap of lenslets is perpetuated at each successive major plane throughout object/image space.

Closely examining the geometry of the light ray representation by studying a small section of the lenslets, in this case an arbitrary five pixels behind each lenslet (Figure 2.7), it can be seen that there are distances of $n+1$ pixels between each pixel intersected by rays originating from any given projection point e.g. the central projection point sends out rays that hit pixels 12, 6, 0, -6 and –12.

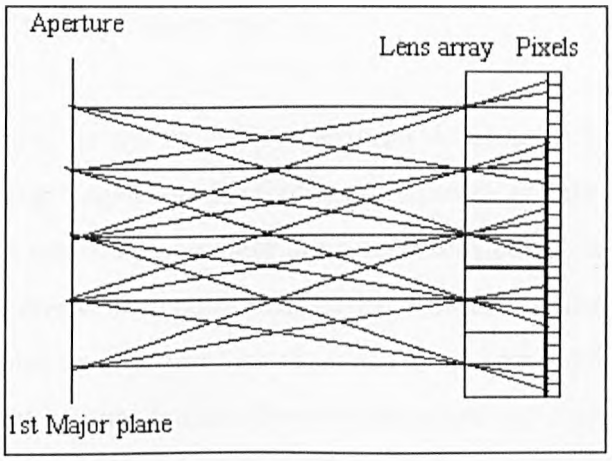

Figure 2.7: Small section of a light ray representation

Parallel rays, however, intersect at every *nth* pixel. In addition, it is seen that there are pixels that are not hit by rays at all.

For the pixels not hit by rays it is obvious that more projection points above and below those already in place are required, or more lenslets. Lens arrays are used, of course, with hundreds of lenslets and therefore these empty pixels will not occur. Another observation is that the capture/replay mesh of rays is so static that only one fixed perspective can exist whatever projection plane location is chosen. If this model were used the number of projection points required would be the same as the number of lenslets and would make real-time generation difficult, although as each projection point only covers relatively few lenslets (as many as pixels behind each lenslet) one might assume that each projection would only take a small amount of processing time. The problem with this model is that three corners of each object triangle are required to be at the image plane simultaneously for interpolative shading. However, at the edges of each projection cone there will be many triangles that will not be able to fulfil this requirement and many expensive clipping procedures would be required. As it stands this model is also not suitable for a forward geometrical projection because unlike integral raytracing the two fixed points that are extrapolated are projection point and object triangle corner.

Options to overcome these problems are:

1. Model the lens array as fully apertured lenslets (i.e. finite-sized lenslet apertures) using transfer and refractive equations. In this case each ray would be allowed to intersect anywhere upon each lenslets surface of curvature. The result is that any object point within the 'volume of freedom' created, from projection point to any point on the circular or semi-cylindrical lens surface, can be imaged by that lenslet. Due to refraction the majority of rays will hit the same pixel as in Figures 2.1 and 2.7 but there will be a spread of pixel hits due to spherical aberration. However, this spread only occurs for object points close to the lens array (as in standard geometric optics) and does not affect the quality of the replayed image, as parts of a scene close to the lens array on replay have the greatest focus. This anomaly is explained in more detail in later chapters. The number of projection points required using the model of Figures 2.1 and 2.7 would still necessarily be the same as the number of lenslets in the lens array and problems will still occur with extensive clipping procedures.

2. Capture 2D perspective view images from the projections of the scene from each projection point and, after rendering each 2D image, select the relevant pixels to remain in a composite LeSD. The relevant pixels would be every $n+1$ pixels from each monoscopic image produced in this way. However, it is noticed that parallel groups of rays are present, each ray in a group is emitted from a different projection point, and there are n numbers of parallel groups. If n 2D orthographic view images were produced, each derived from its own angle relative to the optical axis, the relevant pixels from each image to remain in a composite LeSD would be every *nth* pixel. The practical difference between these two compositing methods, if the model of Figures 2.1 and 2.7 is strictly adhered to, is that the first requires the same number of 2D images as lenslets, where each projection point has a fixed angle of the field of projection and the 2D images are only parts of the scene and the second requires only as many 2D images as there are pixels behind each lenslet i.e. n numbers of 2D images, and the 2D images are the same size as the lens array.

An integral imaging *raytracing* program has no problem with the model of Figures 2.1 and 2.7, as each extrapolated trace from pixel through pinhole only requires it to recognize an intersection with an object algebraically and eventually return with an intensity value.

It can be seen that production of forward geometric integral imaging can be achieved by modifying well known complex standard graphics renderers or decoder/renderers or taking a multiple of their 2D image outputs for compositing, and as perspective imaging is the norm reliance on applications that enable the orthographic solution, as in the second method of (2), is not advisable.

2.2.4 Variable perspective

A further problem encountered using the capture model of Figures 2.1 and 2.7 is that only a fixed perspective is presented. Whatever the position of the projection point plane the perspective within the scene will remain constant. This is because the playback mesh of Figures 2.1 and 2.7 is fixed, and there is nothing that can easily change this. However, the spatial information originally captured within the LeSD can be changed, that is perspective could alter for each change in the projection point plane. The objective could be to modify the capture process to allow this to be built in to the LeSD.

Figure 2.8 portrays two lines of equal dimensions but at different depths in object space. The lines are projected from a point on an aperture or projection point plane and captured at the image plane. The captured lines assume different dimensions due to perspective. If the distance between the aperture and image plane is increased as in Figure 2.9, perspective changes as in real life, and the change between perspective A and perspective B can be compared (bottom of Figure 2.9). The perspective change corresponds to real life, since the further an observer is from a scene the smaller the scene appears to be. Neither a change in perspective, nor a change in the size of the scene, is inherent within the static model of Figures 2.1 and 2.7. Finally, in standard computer graphics, object files allow the user to set the distance to the scene; the further from the scene the more of the scene can be viewed. With integral imaging, if the scene is pushed further away from the aperture then it will be pushed more and

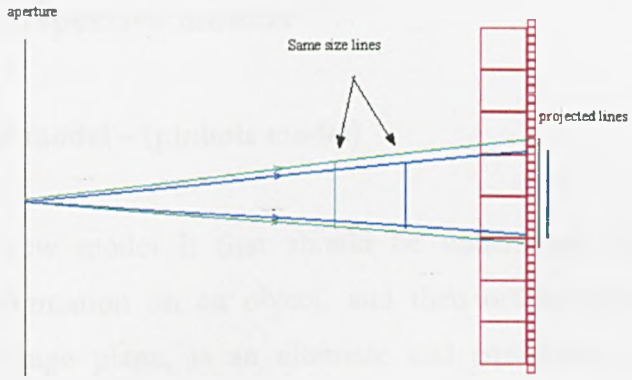

Figure 2.8: Perspective A

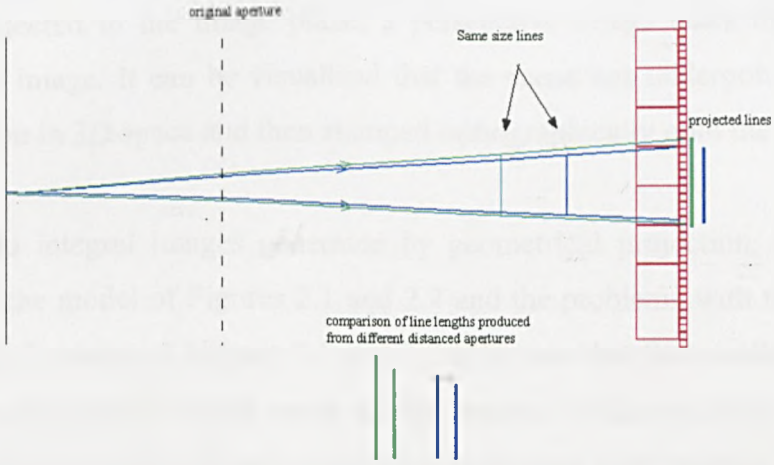

Figure 2.9: Perspective B

---

more behind the lens array, but as the optimum depth within a scene, to position a lens array in close imaging, is approximately 33% [2], this optimum depth will not always be possible. Therefore control over the distance to the scene must be achieved by moving the aperture (called the camera in standard graphics). This natural control is not possible, using the model of Figures 2.1 and 2.7, as the scene remains the same size wherever the aperture is positioned; although it would be possible to artificially scale down the size of the scene. The new model therefore requires that perspective projection be used to give variable perspective and scene dimension change when moving the aperture. Variable perspective models are therefore required.

2.3 Variable perspective models

2.3.1 3D-from-2D model – (pinhole model)

To achieve the new model it first should be understood that by performing a perspective transformation on an object, and then orthographically projecting the result onto the image plane, is an alternate and equivalent approach to directly projecting the object from a projection point to the image plane [50]. Therefore if a projection point is positioned at the centre of the parallel orthographic lines and the scene is projected to the image plane, a perspective image takes the place of the orthographic image. It can be visualized that the scene has undergone a perspective transformation in 3D space and then stamped orthographically onto the image plane.

In relation to integral images generated by geometrical projection, this solves the problems of the model of Figures 2.1 and 2.7 and the problems with the first method in option (2). Looking at Figures 2.1 and 2.7 it is seen that the parallel lines used to produce an orthographic image reach all the lenslets within an array. Similarly, the spread of rays from each correctly positioned projection point reaches the whole lens array. Consequently, each group of parallel lines on replay will now contain variable perspective and variable scene size information and this variance will depend on the position that the projection point plane was set at during capture. Although the replay geometry is fixed the information being replayed is captured using variable geometry.

The field of projection of each orthographically centred projection point has increased to the whole array and by accommodating this the imaginary barrier set up at the interface of the lenslets can be eliminated. The number of projection points required for the first method of option (2) is now the same as the number of orthographic projections of the second method in option (2) i.e. n.

A method to produce integral images with variable perspective by the use of 2D images captured at the image plane is now outlined:

- Position the projection points at the central position of each group of parallel rays, as in Figures 2.1 and 2.7, on an aperture
- Each projection point's field of projection is to include the whole lens array but not captured via a lens array
- Capture 2D images at the image plane as if projected from each projection point
- Each 2D image is a sub-image from which every *nth* pixel is selected for the composite LeSD
- The first of each *nth* pixel from each 2D image is incremented by one for each image in turn

2.3.2 Lens array model - (finite-sized aperture model)

If the barriers between the lenslet interfaces is brought down for the finite-sized aperture method of option (1) the number of projection points required for any given depth can be derived by first looking at the definition of an integral image. The main attribute that separates integral imaging from other 3D display techniques is that the image when viewed has seamless all round viewing. It is not discretised into planes with the accompanying cardboard cutout effect as with multiview techniques. This is accomplished by imaging every point in the scene in adjacent lenslets at least once, the deeper the point the more lenslets image that point. By knowing the maximum depth in the image a calculation can be performed to give the spacing between and thus the number of projection locations to guarantee that all parts of the scene will be imaged in adjacent lenslets (see Figure 2.10). Parts of the scene increasingly greater than z will exhibit a noticeably increased flipping effect because these will not be imaged in adjacent lenslets, whilst those closer to the lens array than z will be increasingly anti-aliased as these points are imaged by their adjacent lenslets more than once. The projection point spacing is calculated from a simple equation, $P(d-z)/z$, which is directly dependent upon the lenslet pitch and indirectly dependent upon the pixel pitch and focal length that are implicit within the calculation of d (Figure 2.6). Previously it has been observed that only at the first major plane is information coming from adjacent lenslets at each intersection junction and it may be thought that only when viewing the display at the first major plane will an image of an

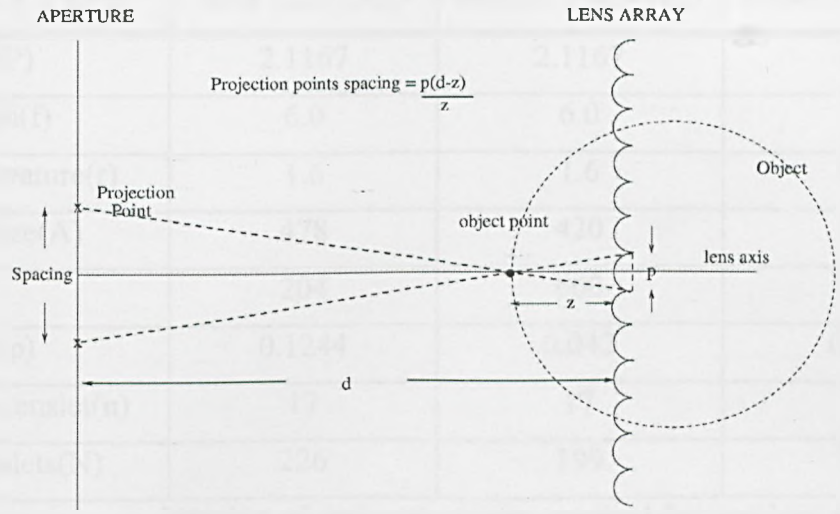

Figure 2.10: Projection location spacing

integral nature be seen. However, if for example, the viewer is positioned at the second major plane, which is approximately twice the distance, the eyes will be receiving a wider view of the display by a factor of 2 and consequently there will again be information from adjacent lenslets.

The model for option (1) now changes from that requiring as many projection points as lenslets (specifically for integral raytracing) to one that requires only those projection points needed for any particular integral imaging system and depth. By using this method it can be seen that the maximum object/image depth of Figures 2.1 and 2.7, that will provide a true integral image, is the mid-point plane between the lens array and projection plane where $z = d/2$. Using the equation from Figure 2.10 confirms the spacing of one lenslet i.e.

$$Spacing = \frac{P(d-z)}{d/2} = \frac{P(d-d/2)}{d/2} = P$$

This is an unlikely scenario for large arrays and present display resolutions due to depth resolution limitations. Examples of the required number of projection points for specific display systems for front of display depths of 40mm, 80mm and 120mm are shown in Table 2.1; all numbers representing dimensions are in mm.

DISPLAY TYPE	IBM T221 LCD	600DPI PRINTED	300DPI PRINTED
Lens pitch(P)	2.1167	2.1167	0.6
Focal length(f)	6.0	6.0	1.7
Rad. of curvature(r)	1.6	1.6	0.16
Aperture Size(A)	478	420	420
dpi	204	600	300
Pixel pitch(p)	0.1244	0.042	0.085
Pixels per Lenslet(n)	17	17	7
No. of Lenslets(N)	226	199	700
Number of projection points required for new lens array model			
40mm depth	9	9	30
80mm depth	11	19	63
120mm depth	17	30	98

Table 2.1: Number of projection points required to generate integral images for three different display types for three different depths in front of the display

Using the capture system of Figures 2.1 and 2.7 would require 226, 199 and 700 projection points respectively, whereas the spacing equation in the lens array model allows for significant processing savings by only using the number of projection points actually required for a given depth.

2.4 Effect of moving the aperture

Figure 2.11 diagrammatically describes the new variable perspective models for option (1) and the second method of option (2). Although the purpose of the new option (1) model is to use less projection points by only using those required for a given maximum system depth, Figure 2.11 can be used to compare with the typical raytracing model of Figures 2.1 and 2.7. The object/image depth necessary to retain an integral nature, in this analysis, is similarly the mid-point plane between the lens array and projection plane.

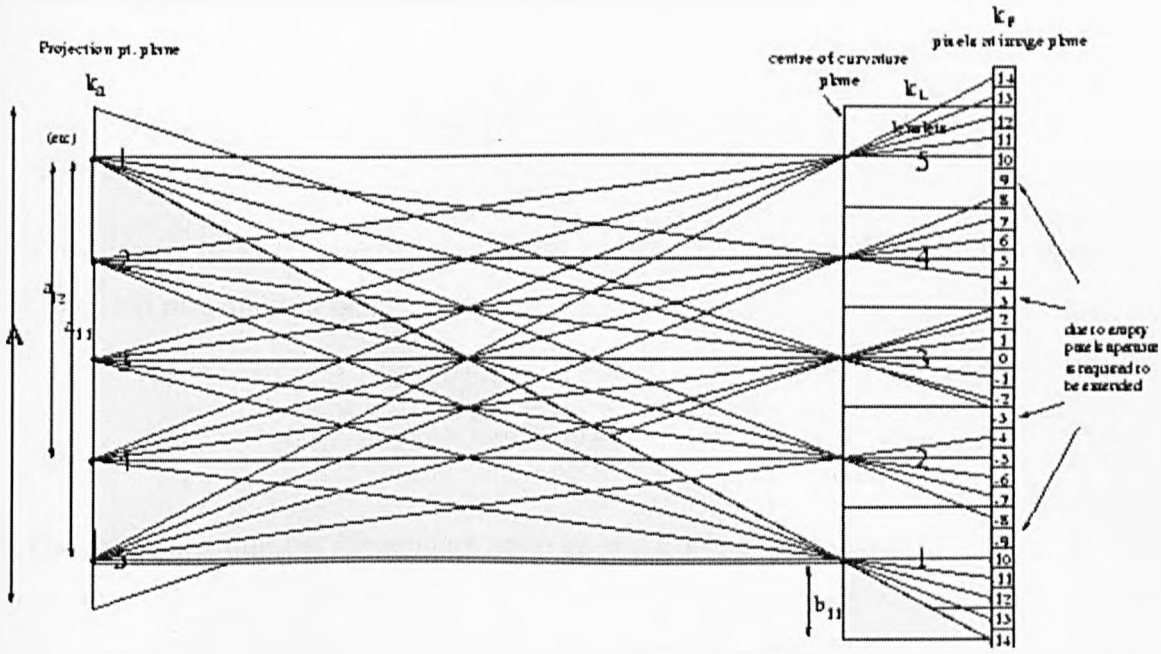

Figure 2.11: Forward geometric, variable perspective model

Immediately it can be seen that the imposed imaginary barrier between lenslets has been removed allowing each projection location the freedom to affect the whole image plane (as in real life when using an aperture). Using the following derived equations, in the following order, the pixel number hit from any ray originating from any radiant can be calculated for any distance of aperture (hence size of aperture) from the lens array.

1) Distance of projection point from the top of the aperture (dependent upon ka):

$$V_{\substack{ka \\ 1->n}} = \frac{Ax(2\ ka - 1)_{1->n}}{2N}$$

2) Vertical distance from the bottom of the aperture to the lenslet pinhole (dependent upon kl):

$$Y_{\substack{kl \\ 1->n}} = \frac{A(x-1) + P(2\ kl - 1)_{1->n}}{2}$$

3) Part of aperture used to calculate b (dependent upon ka and kl):

$$a_{\substack{kakl \\ 1->n}} = xA - V_{\substack{ka \\ 1->n}} - Y_{\substack{kl \\ 1->n}}$$

4) The distance of a pixel hit from its parent lenslet mid-point pixel (dependent upon ka and kl) measured in pixels:

$$b_{\substack{kakl \\ 1->n}} = \frac{-a_{\substack{kakl \\ 1->n}} n}{xA}$$

5) The final pixel number (dependent upon ka and kl) - not rounded off:

$$kp_{\substack{kakl \\ 1->n}} = -\left[\frac{n(N - 2\,kl + 1)}{2} + b_{\substack{kakl \\ 1->n}}\right]$$

where:

P = lenslet pitch N = number of lenslets n = number of pixels per lenslet

kl = lenslet number reading from the bottom of the array

ka = projection point number reading from the top of the projection point plane

kp = pixel number relative to the pixel number zero set at centre of array

A = aperture dimension that is the same as the lens array dimension

x = variable multiplier of distance d

The data given in the following tables are calculated from a program (see Appendix B) that was written to tabulate all the pixel hits for either all the projection points to one lenslet or one projection point to all lenslets, using the above equations.

The data using 5 lenslets (as in Figure 2.11) are not presented in a table because a thorough examination using more lenslets is presented. However, they show that as the aperture moves further from the lens array the rays begin to converge towards the central zero pixel e.g. for an aperture distance of xd where $x = 1$, in Figure 2.11, a

pixel span is –10 to –14. For an aperture distance of xd where $x \to \infty$ the span migrates to pixels –8 to –12. The ray mesh now displays the same geometry as that in

Lenslet Number(kl)	Range of Hits		Central pixel	Lenslet span
	x=1	x=∞		
11	25.0 - 29.55	22.73 - 27.27	25	23 - 27
10	19.55 - 24.09	17.73 – 22.27	20	18 - 22
9	14.09 - 18.64	12.73 – 17.27	15	13 - 17
8	8.64 - 13.18	7.73 – 12.27	10	8 - 12
7	3.18 - 7.73	2.73 – 7.27	5	3 - 7
6	2.27 - –2.27	-2.27 – 2.27	0	-2 - 2
5	-3.18 - –7.73	-2.73 - -7.27	-5	-3 - –7
4	-8.64 - –13.18	-7.73 - -12.27	-10	-8 - –12
3	-14.09 - –18.64	-12.73 - -17.27	-15	-13 - –17
2	-19.55 - –24.09	-17.23 - -22.27	-20	-18 - –22
1	-25.0 - –29.55	-22.73 - -27.27	-25	-23 - -27

Table 2.2: As the aperture moves to infinity the mesh increasingly becomes like that of Figures 2.1 and 2.7

Figures 2.1 and 2.7. The further the aperture moves from the lens array the less perspective is present and as $x \to \infty$ there is no perspective at all. This conclusion states that if a capture method is used as in Figures 2.1 and 2.7 then the previously mentioned *fixed* perspective nature of the resulting integral image will be orthographic.

Due to the small number of lenslets in Figure 2.11 empty pixels are present. Using more lenslets resolves this, and the range of values of hit pixels in each lenslet are tabulated for easier analysis than would be possible extracting the same information from a complex mesh figure. Table 2.2 is a system with the same parameters as those in Figure 2.11 with the exception that there are 11 lenslets. The data compares the range of hits for the aperture set at the same size as the lens array ($x = 1$) and the aperture set at infinity ($x = \infty$). The lenslet span is the span of pixels immediately behind each lenslet. The results again show that as the aperture moves towards infinity the pixel number hits migrate to those shown in the lenslet span column hence displaying the same geometry as that in Figures 2.1 and 2.7.

Considering the practical use of arrays e.g. the IBM T221 LCD integral imaging system shown in Table 2.1, it can be seen that 226 lenslets are required within the array to fit the LCD display area. A question that arises is "do pixels that receive hits in an adjacent lenslet to their parent lenslet also receive hits via their parent lenslet?" If the answer were 'yes' then it would be a situation where one pixel would send its intensities in two separate directions, one through the parent lenslet and one through the adjacent lenslet. This would lead to double imaging, and is the reason why NHK decided to use optical barriers between lenslets [30]. Looking at Table 2.2 provides the answer but analysing a given mid-range lenslet (of an array consisting of 226 lenslets) and one of its adjacent lenslets can provide the answer for a real system. Using the program in the mode of 'all projection points and one lenslet (at a time)' and using a suitable lenslet i.e. lenslet number 56, gives values for the pixel hits from each projection point through its pinhole. These values in terms of fractional pixels range from pixel numbers –964.84 to –981.77 where the first 65 projection points (out of 226) cross the lenslet interface and hit pixels at the image plane in the adjacent lenslet (lenslet number 55). The central pixel number of lenslet number 56 is –969 and the pixel numbers that are directly behind the lenslet are –961 to –977.

The same program was run for lenslets 55 and 54 that received successive incursions in turn and the results shown in Table 2.3 can be analysed:

Lenslet Number(kl)	Range of Hits(x=1)	Central pixel	Lenslet span
56	-964.84 to -981.77	-969	-961 to -977
55	-981.92 to –998.84	-986	-978 to -994
54	-999.00 to –1015.92	-1003	-995 to -1011

Table 2.3: Ray incursions into adjacent lenslets do not have hits via the parent lenslet

It can be seen from the range of hits in these three lenslets that there is no overlapping (this can also be seen in Table 2.2) and the same result is propagated through the array. The intensities, which have been attributed to pixels in an adjacent lenslet, replay out of the scene boundaries when replaying through their own lenslet.

It can be seen in Figure 2.11 that pixels 14, 13, 8, -8, -13 and –14 on replay play back through an adjacent lenslet and are within the bounds of the imaged scene. The bounds depend on the aperture size and distance from the lens array. This is another reason to use this technique where the imaginary opaque wall between lenslets is eliminated. When using Figures 2.1 and 2.7 as a capture method, pixels will play back through adjacent lenslets and take part in reforming the scene, but as the pixel intensities were only derived through the pinholes of their parent lenslet, then many ray bundles will be incorrectly addressed. This is an important point because it is generally accepted that side lobes are generated purely by pixels replaying through adjacent lenslets, but this is not the case.

Integral raytracing can also benefit from integral perspective imaging by using the program off-line or incorporating it within the raytracing code to calculate the correct geometry (i.e. correct pixel through correct lenslet) for any required position of aperture. The variable geometry consists of two parts each being the mirror image of the other, the 'mirror' being a horizontal line from the central pixel of the central lenslet to its pinhole.

2.5 Conclusions

Extrapolating rays from pixels through their parent lens array pinholes and out into object/image space creates a simplification of the way a lens array replays a pixelated surface. To use this playback model as a capture model for a forward geometric projection technique, to produce integral images in real-time, is fraught with difficulties, the greatest being processing time and lack of perspective. By analysing the mesh, two possible options have been outlined that overcome some of these problems. These options were then developed into methods that overcame all the problems associated with the simplified playback mesh of Figures 2.1 and 2.7. The methods allow variable perspective to exist in replayed images by eliminating the virtual wall between lenslets and thereby allowing projection points to 'see' the whole lens array. This in turn enables less numerous projection points to be required by both methods.

For one method - the lens array model - the number of projection points to produce integral images is based on the depth of scene. The spacing between projection points, and hence the number of projection locations to provide a LeSD, can be derived by simple equations. The other method – the 3D-from-2D model - makes use of perspective projections in the place of the equivalent orthogonal projections by positioning the projection points at the centres of the parallel orthographic groups and compositing the numerous 2D images. The number of parallel groups, hence the number of 2D images, is the same as the number of pixels behind each lenslet.

The following two chapters initially describe simple methods for producing spot or mesh integral images to ascertain that volumetric images can be produced by a forward projection method. The two chapters after these directly adhere to the new models explained in this chapter for the *rendering* of integral images with the advantage of the experience gained from the spot or mesh methods.

3

Forward Projection Pinhole Mesh Model

3.1 Introduction

The initial aim is to verify that computer generated integral images can be produced using forward geometric projection. It has already been shown that computer generated integral imaging is possible using a raytracing technique, though each point in the scene is not actually imaged in adjacent lenslets. This is due to the geometry of the ray tracing technique whereby the rays are extrapolated from a pixel through a pinhole, and requires considerable anti-aliasing to closely approximate true integral distributions. The forward projection of the object or scene can overcome this problem by directly taking the triangular points of each object (assuming the scene is represented by triangular mesh data) directly through the adjacent lens array pinholes, thereby guaranteeing a continuum of views in the final integral image. This technique also has the advantage, as in standard computer graphics, of only using processing time when dealing with actual objects, and not wasting processing time in searching for object intersections as in ray tracing techniques, hence enabling real-time generation whereby loss in quality is made up for in speed.

This and the following chapter describe techniques to produce integral *mesh* images, which are the simplest types of images to produce. Initially, a technique using a simple pinhole model of the microlens array is examined (Chapter 3), and subsequently a method of producing integral mesh images using a finite-sized aperture model for each lenslet in the array is considered (Chapter 4). These minimalist, non-rendering programs should give clues to the problems that might later be encountered in developing a more complex rendering system based on the conclusions of Chapter 2.

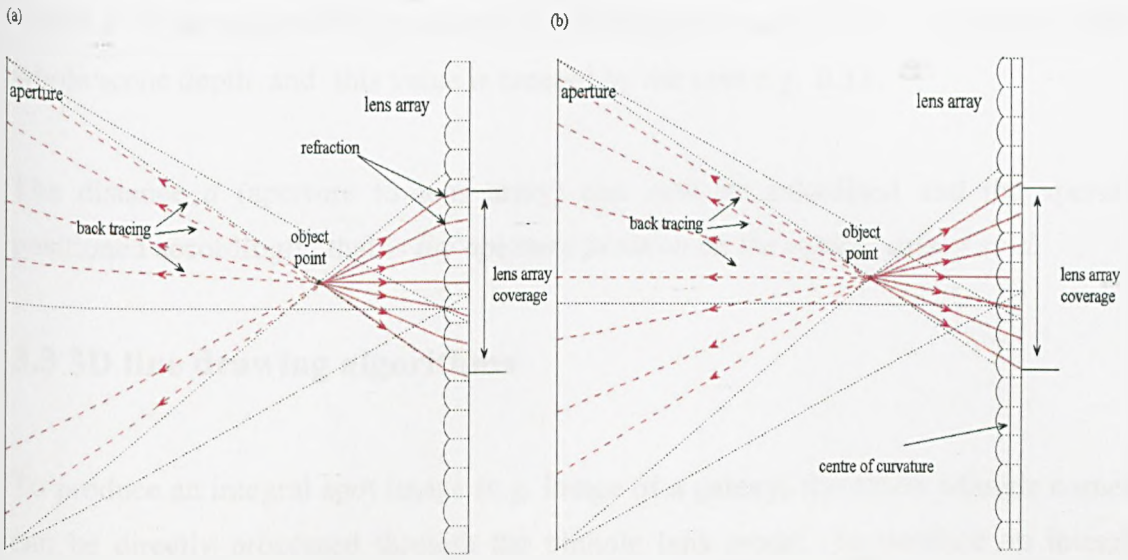

Figure 3.1: Pinhole model used in producing integral images with uni-directional parallax (a) rays intersecting lenslet vertices (b) rays intersecting lenslet centres of curvature

3.2 Aperture and lens array positioning

The vertex of each lenslet in a lenticular and microlens sheet is modelled as a pinhole. The object, in this simple model, is not directly projected from an aperture or projection plane along with equally spaced radiants. This is because the lines extrapolated in that case would be from projection points through object points and would not pass through any pinhole. Therefore the object points are directly imaged through their allotted span of pinholes and checked by back extrapolation to ensure that the ray originates from the aperture (Figure 3.1). If not, then that pinhole is not part of the lens coverage for that object point.

To find the lens coverage for each point, and position the virtual lens array within the scene, it is first necessary to know the maximum and minimum z-coordinates of the scene (*max_zv and min_zv*) where *z* is the optical axis. This must be performed after any scene scaling requirements. With this information the encoding lens array can be positioned at a given depth within the scene

$$zv = (zv\_pos \times (max\_zi - min\_zi)) + min\_zi$$

where *zv* is the z-coordinate position of the lens array and *zv_pos* is a fraction of the whole scene depth and this value is entered by the user e.g. 0.33.

The distance *d* (aperture to lens array) can now be calculated and the aperture positioned accordingly, that is *the aperture position on the optical axis = zv-d.*

3.3 3D line drawing algorithms

To produce an integral spot image (e.g. image of a galaxy) the object triangle corners can be directly processed through the pinhole lens model. To produce an integral mesh image the perimeters of all object triangles must be present in the final image. In standard computer graphics the pixel hits are calculated for the triangle corners and a fast 2D line-drawing algorithm is all that is necessary to draw the perimeters. However, to capture the correct spatial information at the image plane, in the case of integral imaging, it is first necessary to draw the triangle perimeters in 3D *object* space before translating the scene to the capture plane.

3.3.1 Optimum spacing between points representing a line

Within any raytracing technique rays or lines are generally defined as:

$$x = x1 + (x2 - x1) * t = x1 + i * t$$
$$y = y1 + (y2 - y1) * t = y1 + j * t$$
$$z = z1 + (z2 - z1) * t = z1 + k * t$$

These equations were put into a loop within the program to generate a sequence of points making up the perimeters of the triangles (see Appendix C, Part 1). The problem with this method of 3D line drawing, when used in a forward projection integral imaging program, is that the distance between the points making up the lines depend on the initial values of the start and end points e.g. *x2* and *x1*. That is the xyz points are not equally distanced from each other for different lengths of lines, and as object triangles are generally of different proportions, too many or too few points will

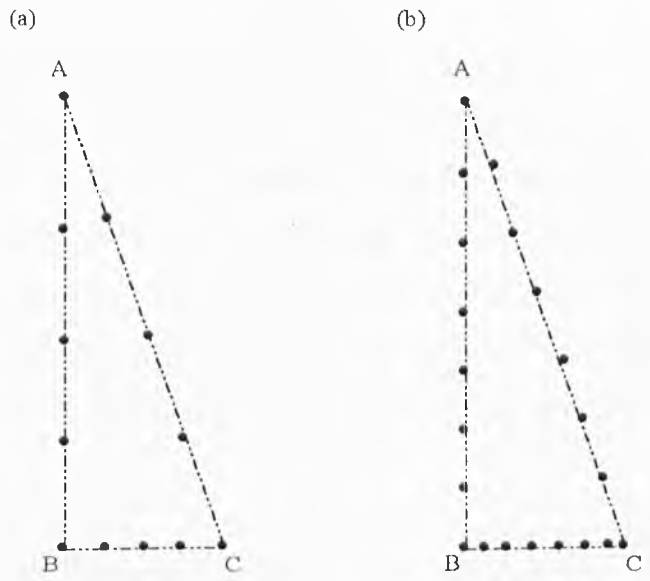

Figure 3.2: Incorrect balance of points making up a line

Figure 3.3: Points representing lines can produce dense areas if the number of points is dependent upon the length of the line

result depending on the lengths of the three lines forming the perimeter. Figure 3.2(a) shows a triangle with the points representing the perimeter and corners. Line BC appears reasonable but the points making up lines AB and AC are further distant from each other and this results in an aliasing problem. Decreasing the value of t increases the point spacing for lines AB and AC (Figure 3.2(b)) but now line BC has more points. Applying this in practice produces undefined clusters of dense areas within an image. This effect can be seen in the right hand of the mesh avatar LeSD in Figure 3.3.

Continually to make changes to the value of t would not be user friendly, and of course not all parts of a scene are affected by aliasing or dense area problems. Therefore a better solution is to produce a 3D line drawing algorithm that generates equally spaced points independent of the line lengths and discover an optimum spacing suitable for integral imaging.

If a line of xyz points, in an object triangle's side, are not parallel to the aperture then the angle subtended at say an adjacent pair of points, with respect to the lens array, is smaller than that subtended by a similar pair that are parallel to the lens array. This means that to image a line correctly in adjacent lenslets by rays originating from the same projection point and hence suffer no aliasing effects requires calculations on a parallel string of points to find the optimum spacing. To err on the side of caution, and pre-empt future technology to allow greater improvements in integral depth resolutions (see section 1.5), a line of points used for the calculation is positioned at the midway position between the aperture and the lens array (see Figure 3.4). They are positioned to symmetrically straddle the optical axis. It can be seen by the use of similar triangles that:

$$\frac{P}{d} = \frac{s}{d/2}$$

$$\therefore s = \frac{P}{2}$$

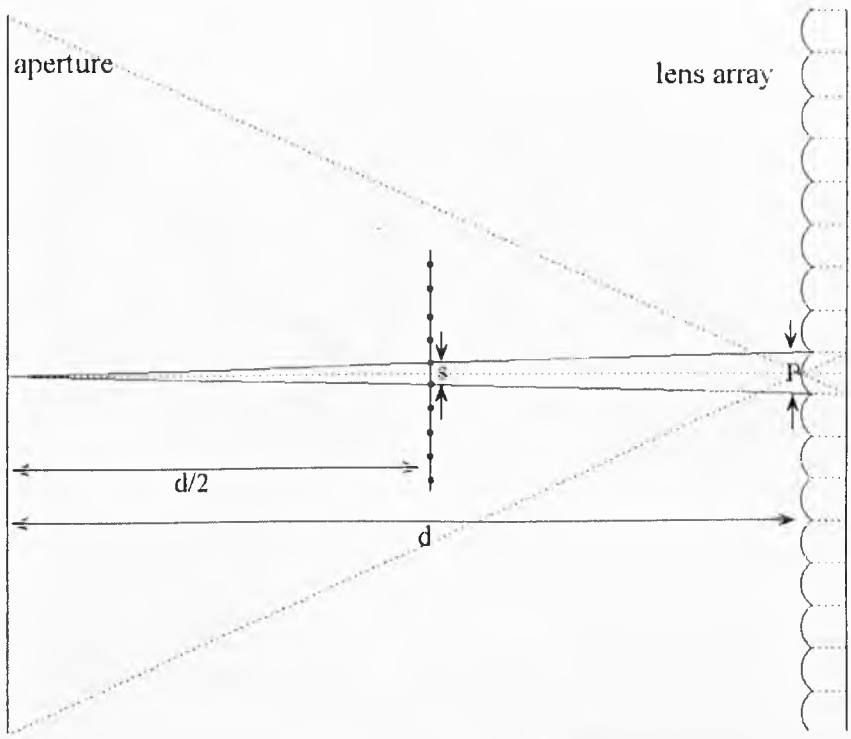

Figure 3.4: Calculating optimum spacing between object points that make a line

This shows that a *safe* spacing between the points that make up the lines within the perimeter of object triangles in 3D space is *P/2.* As the lenslet pitch information will always be supplied by the user the spacing can be calculated within the program and remain transparent to the user.

3.3.2 Producing lines of equidistant points in 3D space

Each side of a triangle exists in 3D space and the direction of any line between two corners can be oriented in any direction. If the line does not lie in the same plane as the x, y, or z axis (i.e.skew) then the orientation of the line can be described by the angular separation from the x and y axis (α, β). However, if the line lies in the same plane as the x, y, or z-axis then only one angle needs to be considered. Figure 3.5 enables the different orientations to be visualised more clearly. The line start points are **x1, y1, z1** and end points **x2, y2, z2**. The distance the line travels across the x-axis is denoted as ***alpha***, y-axis as ***beta*** and z-axis as ***delta***.

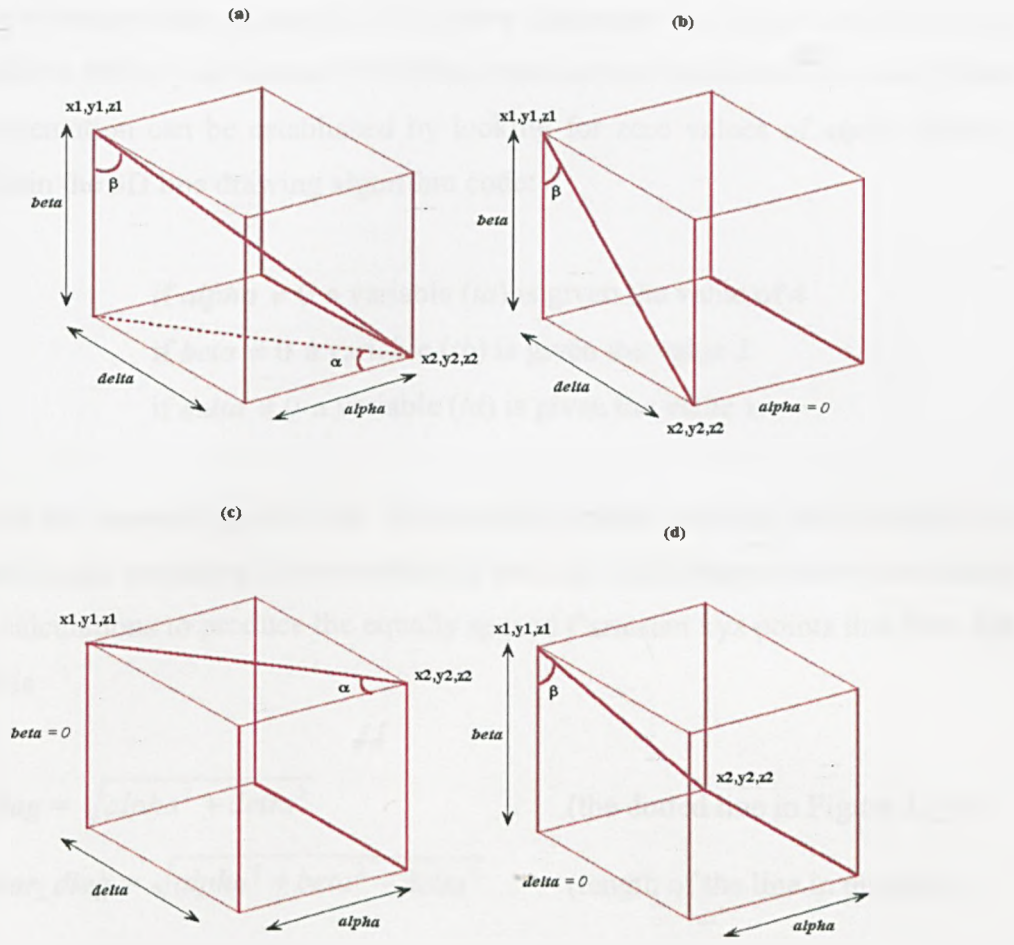

Figure 3.5: (a) skew line (b) line in z-axis plane (c) line in x-axis plane (d) line in y-axis plane

---

(a) Shows a skew line crossing all axis planes

(b) Shows a line where ***alpha*** = *0*

(c) Shows a line where ***beta*** = *0*

(d) Shows a line where ***delta*** = *0*

To first calculate the displacement of the line from the three axes i.e. ***alpha***, ***beta*** and ***delta***, it is only necessary to subtract each end-point from the start-point, that is

$$\begin{aligned} \textbf{\textit{alpha}} &= \mathbf{x2 - x1} \\ \textbf{\textit{beta}} &= \mathbf{y2 - y1} \\ \textbf{\textit{delta}} &= \mathbf{z2 - z1} \end{aligned}$$

Three more orientations are possible if any two displacements equal zero, these lines run parallel to the x, y or z axes. Therefore, there are seven orientation possibilities and the orientation can be established by looking for zero values of ***alpha***, ***beta*** or ***delta***. Within the 3D line drawing algorithm code:

if ***alpha*** $\neq 0$ a variable (*ta*) is given the value of 4
if ***beta*** $\neq 0$ a variable (*tb*) is given the value 2
if ***delta*** $\neq 0$ a variable (*td*) is given the value 1.

When these are summed together the values are any integer between, and inclusive of, 1 - 7, each integer revealing the orientation of the line. Each integer value necessitates different calculations to produce the equally spaced Cartesian xyz points that form the line. That is

$\boldsymbol{planar\_diag} = \sqrt{alpha^2 + delta^2}$ (the dotted line in Figure 3.5(a))

$\boldsymbol{non\_planar\_diag} = \sqrt{alpha^2 + beta^2 + delta^2}$ (length of the line in question)

ta + tb + td	α	β
7	$\tan^{-1}$ (delta/alpha)	$\tan^{-1}$(planar_diag/beta)
6	0	$\tan^{-1}$(alpha/beta)
5	$\tan^{-1}$ (delta/alpha)	0
3	0	$\tan^{-1}$(delta/beta)

Table 3.1: Angles of line displacement

Numbers 1, 2 and 4 are not included in Table 3.1 because they have two displacements equalling zero, hence there are no angles to calculate. Now that the required angular separations for the line in question have been calculated the line points may be calculated within a program loop that starts at *P/2*, and is incremented by *P/2* whilst $n \leq$ ***non_planar_diag*** (*n* is the variable that holds the resulting length after each increment). Hence each increment along the line produces a new set of points (*dx, dy, dz*) as follows in table 3.2

ta + tb + td	dx	dy	dz
7	n*cosα*sinβ	n*cosβ	n*sinα*sinβ
6	n*sinβ	n*cosβ	0
5	n*cosα	0	n*sinα
4	n	0	0
3	0	n*cosβ	n*sinβ
2	0	n	0
1	0	0	n

Table 3.2: Calculations for incrementing points along a line

The end-point must also be included because the line length ***(non_planar_diag)*** when divided by *P/2* will rarely produce a remainder of zero.

The line drawing function is incorporated into the integral renderer and works very well (see Appendix C, Part 2), but the code, although accurate, is not very elegant. However, on closer examination of Table 3.2 and Figure 3.5(a) it can be seen that:

row 7, column dx

$$dx = n * \cos\alpha * \sin\beta$$

but

$$\cos\alpha = \textbf{\textit{alpha/planar\_diag}}$$

$$\sin\beta = \textbf{\textit{planar\_diag/non\_planar\_diag}}$$

therefore

$$dx = n * \textbf{\textit{alpha/ non\_planar\_diag}}$$

similarly in row 7, columns *dy* and *dz*

$$dy = n * \textbf{\textit{beta/ non\_planar\_diag}}$$

$$dz = n * \textbf{\textit{delta/ non\_planar\_diag}}$$

Where ***alpha***, ***beta*** and ***delta*** are divided by the length of the line in question, forming the direction cosines **L**, **M** and **N** respectively. By multiplying the direction cosines by a given length (in this case the recursively added 'space between points' value *n*) the new xyz components that form the points along the line result. The calculation for the

line displacement angles are not required as shown in Table 3.1, and neither are the time consuming sines and cosines shown in Table 3.2. Implementing these relationships results in a more concise algorithm (Appendix C, Part 3) that runs much faster than either of the two previously devised 3D line drawing algorithms.

For experimental purposes it is easy to modify the 3D line drawing code, using direction cosines, for either generating the 'space between points' by the dependency upon line length as in the first algorithm (Part 1), or the 'same space between points' whatever the length of line (Part 2). For the former, a pre-loop line of code is required to provide the number of points that are to be generated for that line i.e. *points[u]=floor(non_planar_diag/space)+1*. The integer u can be 0, 1 or 2 representing the three sides of a triangle, and the number of points making up each line are stored in the array *points[u]*. The addition of 1, on the right hand side, is simply that when a line is divided by a number the answer is the number of spaces *not* the number of points. When the total number of points for the first line has been reached in the iteration process the next two lines of the triangle perimeter are similarly dealt with. For the latter, the loop parameters are those for Part 2 as previously detailed. Within the program all xyz points making up the perimeter of a triangle are stored in *lines[i]*.

3.4 Lens array coverage of points at different depths

Given that the aperture size and distance from the lens array is commensurate with the requirements of the accurate image field alignment of the central lenslet, it can be seen that the depth of image point dictates the lenslet coverage for that point (see Figure 3.6). The deeper the point the more lenslets image that point - depth being interpreted as the distance of a particular point to the lens array whether that point be in front or behind the lens array.

3.4.1 Calculating the coverage

To calculate the lens array coverage for a given point is simple, however it does depend on the position of the point relative to the screen plane and can be defined by:

Zo<Zv (points in front) ***Zo>Zv (points behind)***

$$reach1 = \frac{(A/2 + Yo)(Zv - Zo)}{Zo - Zap} \qquad reach3 = \frac{(A/2 - Yo)(Zo - Zv)}{Zo - Zap}$$

$$reach2 = \frac{(A/2 - Yo)(Zv - Zo)}{Zo - Zap} \qquad reach4 = \frac{(A/2 + Yo)(Zo - Zv)}{Zo - Zap}$$

$$reach5 = \frac{(A/2 + Xo)(Zv - Zo)}{Zo - Zap} \qquad reach7 = \frac{(A/2 - Xo)(Zo - Zv)}{Zo - Zap}$$

$$reach6 = \frac{(A/2 - Xo)(Zv - Zo)}{Zo - Zap} \qquad reach8 = \frac{(A/2 + Xo)(Zo - Zv)}{Zo - Zap}$$

The difference in geometry for points in front compared with points behind the lens array is quite clear from Figure 3.6. A point in front of the lens array will, due to the aperture, cover a larger area of the lens array than a point of equal depth behind the lens array. This is what would be expected for the generation of perspective images through an aperture. The rays projecting points that are positioned in front of the lens array cross over, whilst those for points behind do not. This is the characteristic difference that separates these two positions and the spatial twin-depth information, captured at the focal plane, enables the integral scene to be viewed seamlessly straddling the lens array. To view two identically dimensioned objects one completely in front of and one completely behind the lens array, for example two spheres, reveals different shapes within each lenticular lenslet when one is compared to the other [51]. These differences are clearly noticeable only along the edges of objects. The edges replaying within each lenslet of the sphere in front of the lens array follow the shape of the circumference of the sphere correctly, while those replaying the sphere behind the array are laterally inverted. This is to be expected as the inverting action of lenses re-invert the crossed over rays for the object in front of the array, and invert the rays for the object behind the array.

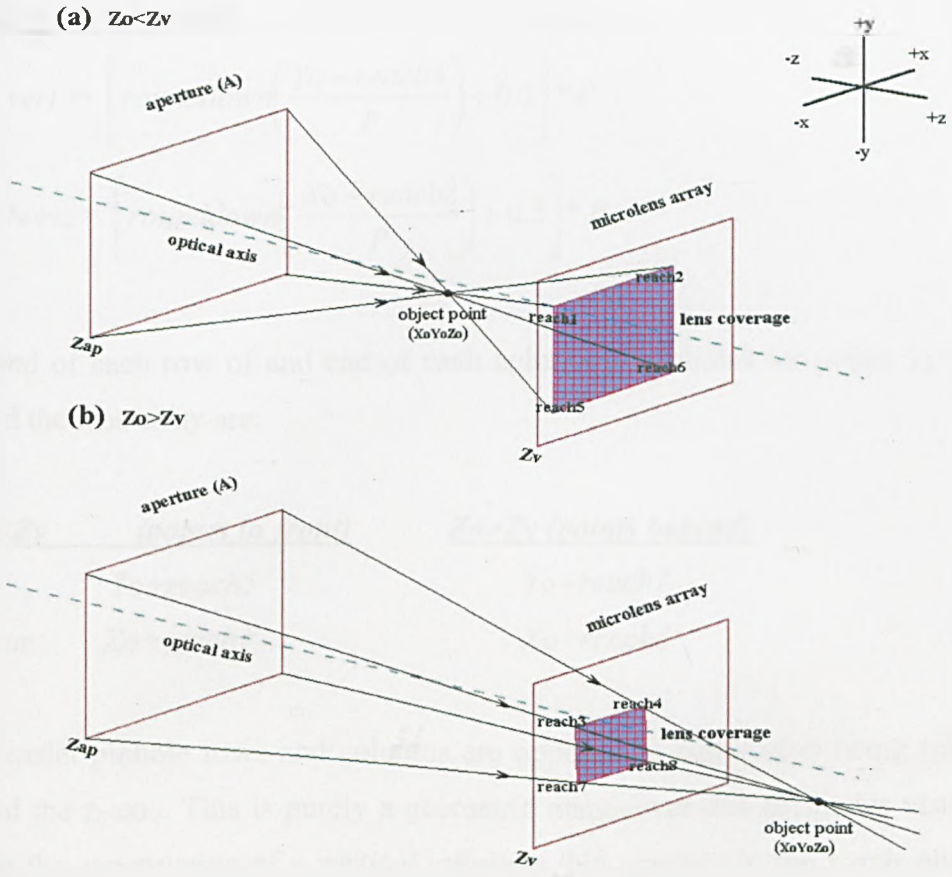

Figure 3.6: The area of lenslets imaging a point is dependent upon the depth of that point and the position of the aperture (a) point in front of array (b) point behind array

---

3.4.2 Finding the pinholes within the coverage

It is now necessary to find the actual lenslet pinholes associated with the *reach* results in order that a double loop can be performed that traverses the lens coverage, stopping at each pinhole coordinate in turn to trace the point in question through it to the capture plane. The positions of the starting pinhole in terms of the y-axis (start_vert) and x-axis (start_horiz) are calculated by first rounding down to a whole number:

Zo<Zv (points in front)

$$start\_vert = \left[rounddown\left(\frac{Yo - reach2}{P} \right) + 0.5 \right] * P$$

$$start\_horiz = \left[rounddown\left(\frac{Xo - reach6}{P} \right) + 0.5 \right] * P$$

Zo>Zv (points behind)

$$start\_vert = \left[rounddown\left(\frac{Yo - reach4}{P} \right) + 0.5 \right] * P$$

$$start\_horiz = \left[rounddown\left(\frac{Xo - reach8}{P} \right) + 0.5 \right] * P$$

The end of each row of and end of each column of pinholes for points in front and behind the lens array are:

Zo<Zv	***(points in front)***	***Zo>Zv (points behind)***
Row:	*Yo+reach5*	*Yo+reach7*
Column:	*Xo+reach1*	*Xo+reach3*

The lenslet pinhole rows and columns are opposite to convention being rotated 90° around the z-axis. This is purely a geometric manoeuvre that facilitates visualisation during the construction of a *vertical* infinitely thin aperture in the *x-axis* plane when replaying integral images with uni-directional parallax (see Figure 3.1). The production of the equivalent equations, and hence code, when using semi-cylindrical lenslets is considerably easier than that required for the circular microlens arrays requiring only half of the equations. The semi-cylindrical equations are already embedded within the 'omni-directional parallax' program (see Appendix D) and as such will not be described here. However, it is worth pointing out that to achieve perspective in x and y directions the infinitely thin aperture must be in a fixed position with respect to the x-axis e.g. $X_{ap}=0$.

Hexagonal lens arrays can also be modelled by a pinhole technique, but two pitches are required as the distance of one pinhole to an adjacent pinhole on the same row is shorter than that to a pinhole in the next row of lenses. Of course, the distance to a pinhole in an adjacent row is in reality a diagonal measurement, but if a vertical measurement is taken each row can be offset from the other, in the double loop, by half the dimension of the horizontal pitch (see Appendix E).

3.5 Capturing object points at the image plane

It is a simple task to trace the rays from object points through the imaging lenslet pinholes, positioned at the centre of curvature, and onto the imaging plane at the focal length of the lens array using similar triangles. To find the centre of curvature it is necessary to calculate the lens radius of curvature and calculate the centre of curvature by subtracting this value from the focal length. That is the focal length through the lens media not the focal length in air. Physically measuring the sag of a lens and using the equation

$$r = \frac{P^2 + 4sag^2}{8sag}$$

where r is the radius, or by using the thin lens equation

$$r = f(n_2 - 1)$$

gives a value for the radius of curvature. Obviously if the focal length (f) is not known then it can be measured by physical means. The material of the lens array allows the refractive index (n_2) to be established.

Alternatively, if the pinholes are situated at the lenslet vertices the use of Snell's Law is required to find the imaging plane and trace intersections, as shown in the equations below:

where: θ_1 is the incident angle of the ray to the lenslet vertex

θ_2 is the refracted angle

xr, yr are the distances of the image plane hit from the lenslet 's vertex

x_fin, y_fin are the image plane coordinates

n_1 is the refractive index of air and is taken to be the value of 1

Zo<Zv (points in front)

$$\theta_1 = \tan^{-1}\left(\frac{Yv - Yo}{Zv - Zo}\right)$$

$$\theta_2 = \sin^{-1}\left(\frac{n_1 \sin\theta_1}{n2}\right)$$

$$yr = f \tan\theta_2$$

$$y\_fin = Yv + yr$$

Zo>Zv (points behind)

$$\theta_1 = \tan^{-1}\left(\frac{Yv - Yo}{Zo - Zv}\right)$$

$$\theta_2 = \sin^{-1}\left(\frac{n_1 \sin\theta_1}{n2}\right)$$

$$yr = f \tan\theta_2$$

$$y\_fin = Yv - yr$$

$$\theta_1 = \tan^{-1}\left(\frac{Xv - Xo}{Zv - Zo}\right)$$

$$\theta_2 = \sin^{-1}\left(\frac{n_1 \sin\theta_1}{n2}\right)$$

$$xr = f \tan\theta_2$$

$$x\_fin = Xv + xr$$

$$\theta_1 = \tan^{-1}\left(\frac{Xv - Xo}{Zo - Zv}\right)$$

$$\theta_2 = \sin^{-1}\left(\frac{n_1 \sin\theta_1}{n2}\right)$$

$$xr = f \tan\theta_2$$

$$x\_fin = Xv - xr$$

Once the coordinates at the image plane are evaluated it is necessary to carry out the pixelation process. The values of *x_fin* and *y_fin* are the 2D coordinate numbers at the image plane and have to be rounded either up or down to the nearest integer value. This is after position adjustments by adding *A/2* to each (to eliminate any negative pixel numbers) and size adjustments dependent upon the display resolution i.e. dpi/25.4. This changes the image stored in the image buffer to the correct size dependent upon the resolution of the output hardcopy or display. The virtual lens array and pixels are now the same size as the real lens array and pixels. When the real array is placed on the top of the LeSD correct registration of the lenslets is possible and the outcome is an integral display (see Figures 3.7 and 3.8). A flowchart of the program makeup can be seen in Figure 3.9.

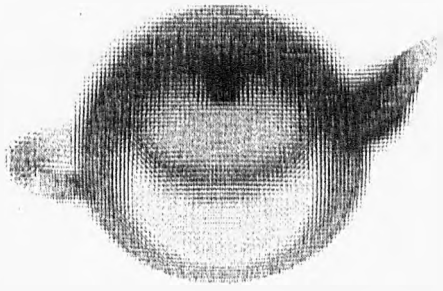

Figure 3.7: Omni-directional mesh distribution generated by the pinhole mesh model

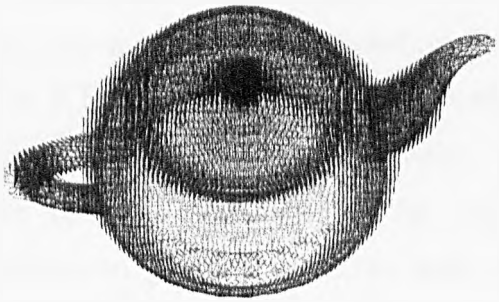

Figure 3.8: Uni-directional mesh distribution generated by the pinhole mesh model

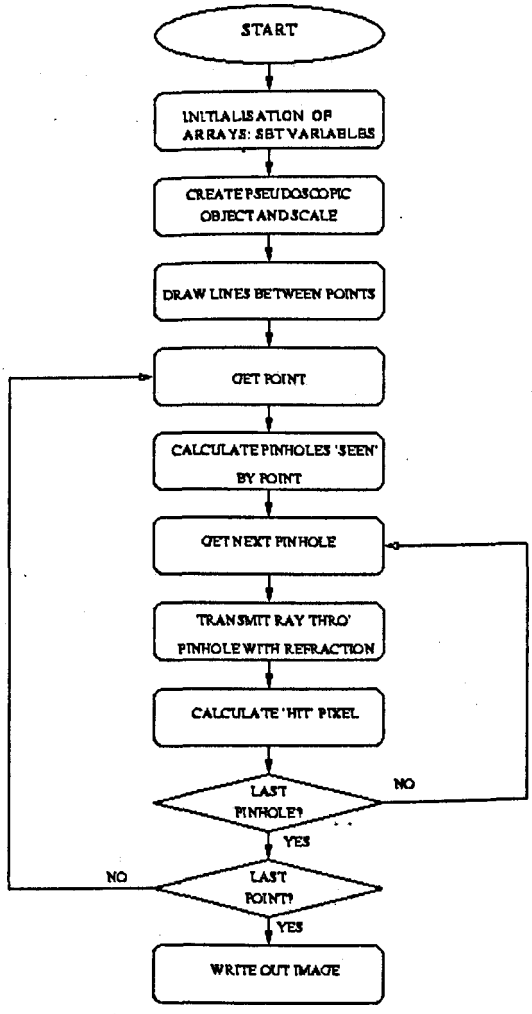

Figure 3.9: Integral pinhole mesh model flowchart

3.6 Conclusions

The production of mesh integral images, albeit using a technique not directly associated with those of Chapter 2, does show that volumetric replication by a forward geometric projection model works. It has revealed problems and solutions to those problems that are required for the forward projection rendering system discussed in later chapters. It has shown that a 3D line drawing algorithm is required to connect the object triangles vertices before translation through the virtual lens array, and produced the optimum equations for the algorithm. Overall, it has explained in detail a method to generate mesh (or spot) integral images using a pinhole technique. To develop a technique that models each lenslet as a finite-sized aperture is the next logical step. In particular it is known that this will introduce

spherical aberration as in a real integral camera, however, the effect on image quality of spherical aberration has not been fully assessed to date. Consequently the aperture mesh model discussed in the next chapter provides an opportunity to simulate such image formation. The work described in this chapter and the next sets the scene for producing *rendered* integral images.

Forward Projection Finite-Sized Aperture Mesh Model

4.1 Introduction

There are problems in standard computer graphics associated with pinhole models [52], and there is no reason to suggest that these problems do not affect integral imaging. Pinholes produce a uniformly perfect focus that eliminates the depth of field associated with finite-sized apertures and this depth of field is a vital cue for depth perception. Another problem associated with pinhole models, and this might include an integral raytracing technique (which is essentially a pinhole model), is that a strobing effect is created when viewing an animation made from a sequence of perfectly sharp images. For the production of integral images, though, the size of replayed points deeper in object/image space that limit the depth resolution may cancel these effects. However, they are more good arguments for the production of computer generated integral images using a finite-sized aperture. A mesh model with optical aberration considerations and the problems of integer or non-integer number of pixels per lens are described in this chapter.

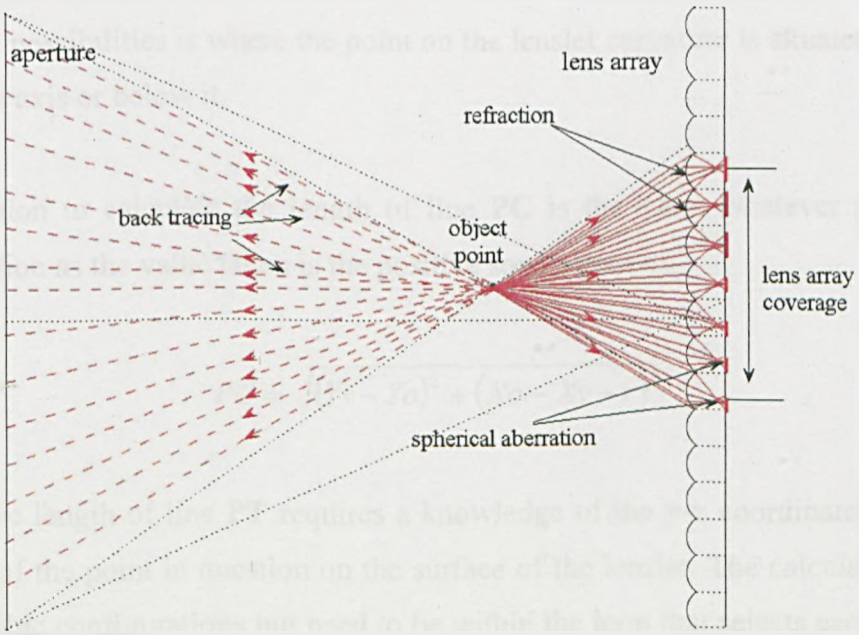

Figure 4.1: Forward projection finite-sized aperture model

4.2 Refraction at locations on the lenslets

To generate fully apertured lenslets by modifying the pinhole model it is necessary to extrapolate rays from each object point to a number of locations on the curved surface of any lenslet within the coverage of each point (Figure 4.1).

From Figure 4.2 the relevant equations describing the behaviour of the model can be established. Arcs are drawn out from the centre of curvature producing points on the lenslet surface. The number of points is input by the user. The distance from point **C** to the chord (i.e. to line **TB**) is first calculated (*c_to_chord*) from which the arc angle is found (*arc_angle*) i.e.

$$c\_to\_chord = \sqrt{r^2 - \left(\frac{P}{2}\right)^2}$$

$$arc\_angle = \tan^{-1}\left(\frac{P/2}{c\_to\_chord}\right)$$

It can be seen that there are many different geometric possibilities to this figure depending on where the object point (**P**) is situated with respect to the optical axis and if it is in front of the lenslet or behind the lenslet. Another variable producing more geometric possibilities is where the point on the lenslet curvature is situated i.e. above the optical axis or below it.

The equation to calculate the length of line **PC** is the same whatever the graphic configuration as the value taken is the positive square root of:

$$PC = \sqrt{(Yv - Yo)^2 + (Xo - Xv - r)^2}$$

To find the length of line **PT** requires a knowledge of the y-z coordinates (*y_chord, z_chord*) of the point in question on the surface of the lenslet. The calculations apply to all graphic configurations but need to be within the loop that selects each curvature point in turn (*nn*) and *arc* is the input variable that is chosen to represent the number

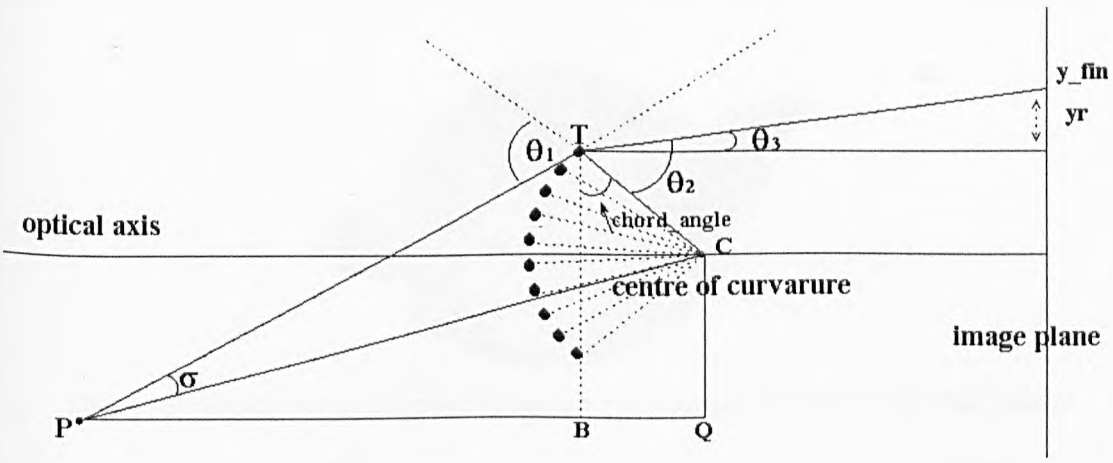

Figure 4.2: Calculations for refracted rays at lens curvature points

of points on the curvature. This number is used as the devisor of the total angle that is swept out by the radius from **CT** to the optical axis and so the total number of points on each lenslet surface is $2arc+1$:

$$y\_chord = Yv + r\sin\left(\frac{arc\_angle(arc-nn)}{arc}\right)$$

$$z\_chord = Zv + r - r\cos\left(\frac{arc\_angle(arc-nn)}{arc}\right)$$

The length of the line **PT** can now be calculated:

$$PT = \sqrt{(z\_chord - Zo)^2 + (Yo - y\_chord)^2}$$

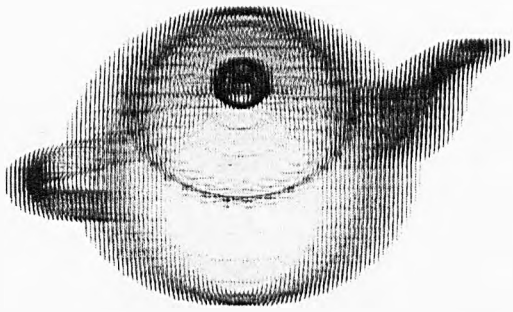

Figure 4.3: Uni-directional mesh distribution generated by finite-sized aperture model

Knowing the lengths of lines **PC** and **PT** the angle σ can be found by the use of the cosine law, from this θ_1 can be calculated, and by the use of Snell's law θ_2 is found:

$$\sigma = \cos^{-1}\left(\frac{PC^2 + PT^2 - r^2}{2PCPT}\right)$$

$$\theta_1 = \sin^{-1}\left(\frac{PC\sin\sigma}{r}\right)$$

$$\theta_2 = \sin^{-1}\left(\frac{\sin\theta_1}{n_2}\right)$$

θ_3 is derived from equations involving the *chord_angle* but because these equations are dependent upon the orientation of the object point to the curvature point they have slight differences from each other. The intersection of the ray at the image plane (*y_fin*) and finally the pixel coordinates in the image buffer (*xf* and *yf*) can now be found.

All the equations for the finite-sized aperture mesh model are used to modify the pinhole model and as such there is an increase in the processing time due to the extra loop. Integral, finite-sized aperture, mesh images produced (Figure 4.3) showed no major objective difference or any noticeable increased degradation when compared to the images generated by the pinhole model. The new flowchart with the extra loop can be seen in Figure 4.4.

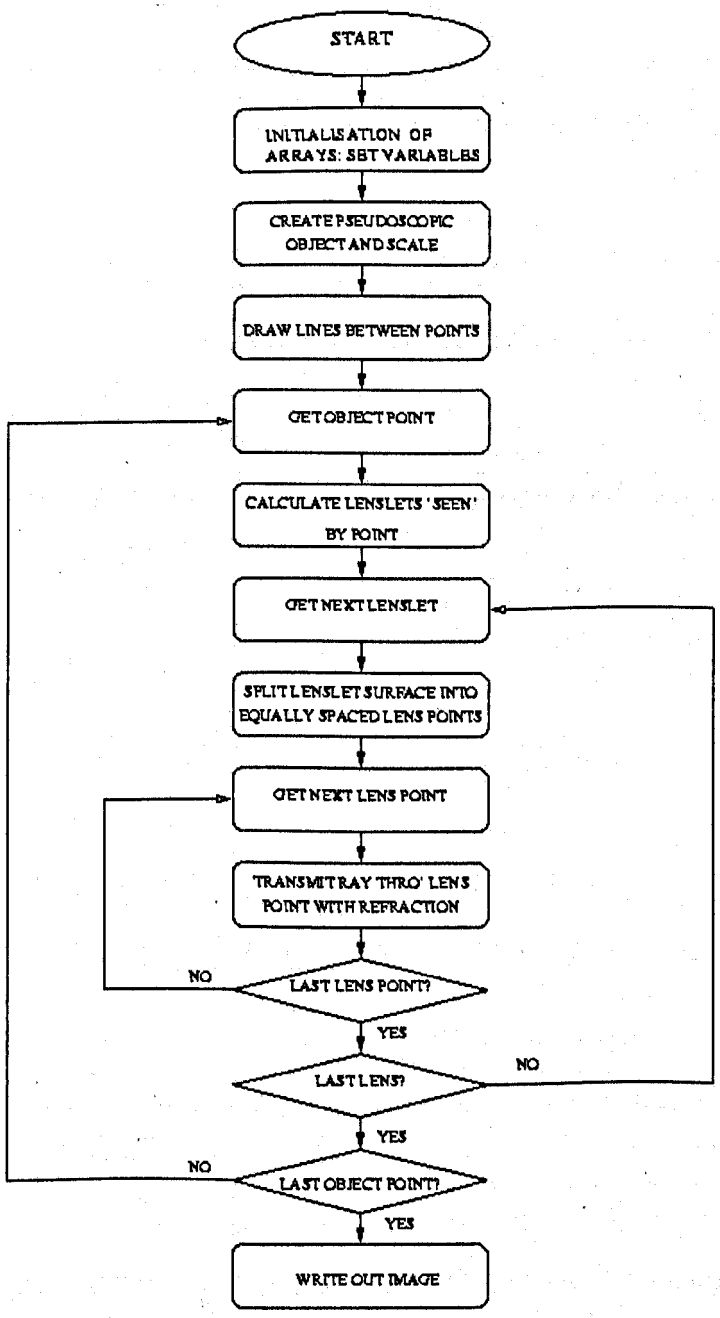

Figure 4.4: Finite-sized aperture model flowchart

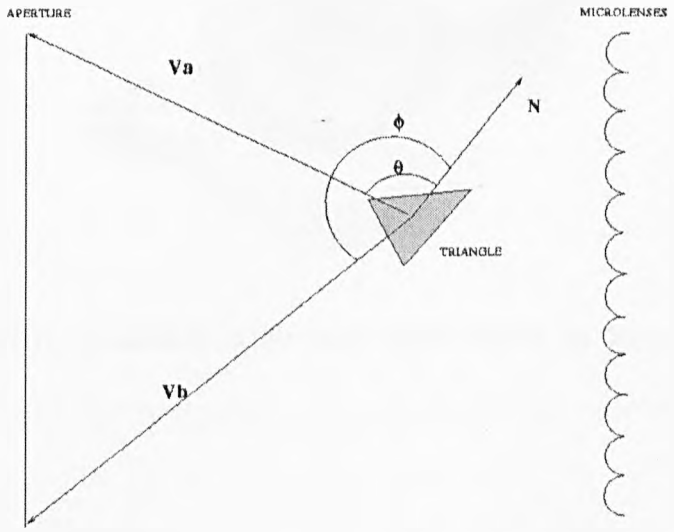

Figure 4.5: Viewpoint for the general cull is the whole aperture

4.3 Backface culling

By testing each triangle for visibility it is possible to introduce backface culling (BFC) with mesh images, this paves the way for the implementation of culling for an integral rendering program. The cross product of two adjacent vectors determine the orientation of each triangle being the vector normal to the plane of the triangle, and this can easily be computed from the cross product determinant. If the triangle is facing away from the viewer it is discarded. BFC is introduced before tracing the rays to the image plane as it eliminates a large percentage of the object data. In the domain of integral imaging the aperture is the viewpoint not a single location. A special type of general culling is therefore necessary that eliminates only those object triangles that cannot be seen from positions within the aperture. The limits are the extremes of the aperture. By constructing the two vectors from each triangle to the aperture extremes ($\mathbf{V}_a$ and $\mathbf{V}_b$) and calculating the angles they make with each triangle normal ($\mathbf{N}$), a general cull can be achieved (Figure 4.5) i.e. the triangle is discarded if (θ and ϕ)>90°. The output can be seen as a backface culled LeSD (Figure 4.6).

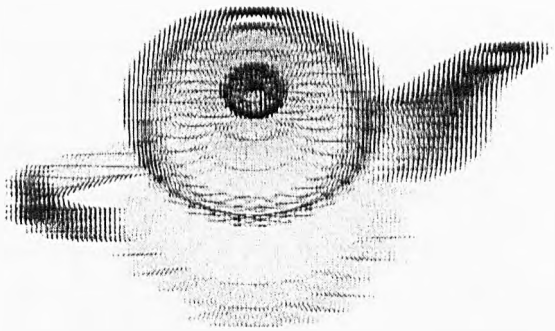

Figure 4.6: Unidirectional culled mesh distribution generated by the finite-sized aperture model

4.4 Optical aberrations

It is well known in the field of optics that lenses and apertures cause image aberrations such as diffraction, spherical aberration, coma, astimatism, curvature of field and distortion. All of these aberrations with the exception of diffraction can be visualised and explained using geometric optics that assumes the straight-line propagation of light in a homogeneous medium. It is geometric optics that is used in this thesis, and in the modelling of the lens arrays only the assumptions inherent in geometric optics are present. Diffraction effects are described by analysing the wave theory of light and this theory deviates from straight-line propagation in direct proportion to the wavelength of light. However, diffraction and how it may or may not affect pixellated integral images is briefly discussed.

4.4.1 Diffraction

Apertures produce Fraunhofer diffraction effects whereby an imaged point consists of concentric bright and dark rings that rapidly diminish in intensity with the increase of radius (see Figure 4.7.)

The point spread is calculated using a form of equation first derived by Airy [53] and is usually calculated to the 1st or 2nd bright ring, further rings are not visible to the naked eye.

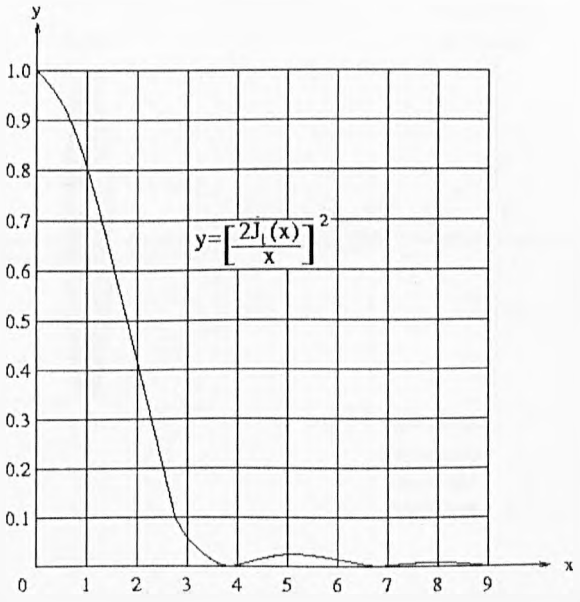

Figure 4.7: Fraunhofer diffraction at a circular aperture

The maximum and minimum radii of the rings of the diffraction pattern is calculated from the following equations:

$$y = \left(\frac{2J_1(x)}{x} \right)^2$$

where $$x = \frac{DP\pi}{2\lambda}$$

P is the pitch of the lens, D is the diameter of the rings, λ is the wavelength of the light.

The 3rd minimum occurs where $x = 3.238\pi$ hence:

$$D = \frac{2 \times 3.238\lambda}{P}$$

It is seen that the point spread is inversely proportional to the lens pitch, so the smaller the pitch the larger is the effective size of the diffraction pattern. The smallest lenticular lens pitch described in this thesis is $0.6mm$. By letting λ be the wavelength mid-range frequency ($500nm$) the value of point spread is found to be $5.4\mu m$. If a hardcopy-printing device, for example, is set at $300dpi$ the pixel size is $0.0847mm$. It can be seen that a single pixel could contain ≈ 246 point spread discs, thus diffraction effects are not a consideration at this resolution and pitch size. Similarly, displaying images upon pixellated

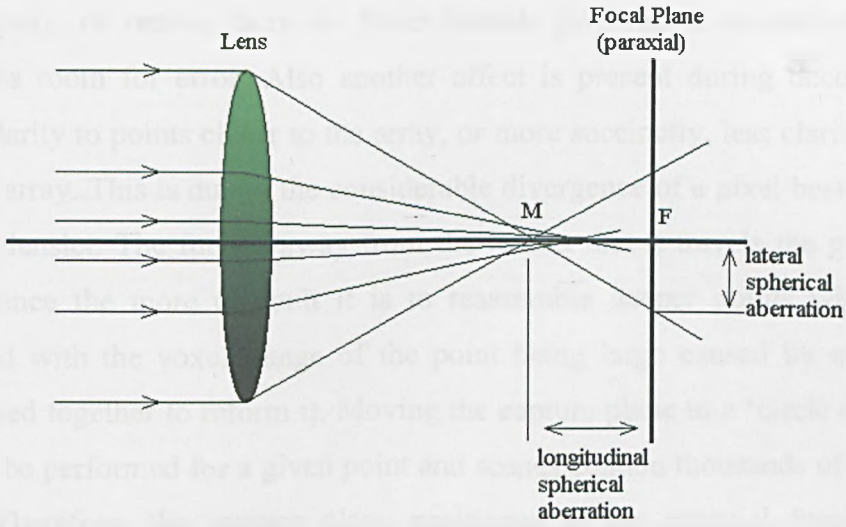

Figure 4.8: Longitudinal and lateral spherical aberration

surfaces such as LCDs have resolution limits far greater than those imposed by diffraction. Hence, when the resolution limit of the detector is much larger than the Airy pattern, the transverse ray aberrations provide a more suitable measure of image quality [54] and as such diffraction effects are not a concern for the capture mechanism used in this research.

4.4.2 Spherical aberration and defocus

Marginal rays intersect the lens surface at the extreme ends of a lens and they tend to be more sharply focused than central rays and this produces spherical aberration (Figure 4.8). The point spread at F is much larger than that at M and so in moving the capture plane to M the effect of the aberration for a given point will reduce. Point M is known as 'the circle of least confusion' [55]. However, points closer to the lens array (within *2f*) have their conjugate focus beyond the paraxial focal plane and hence the rays do not cross over and defocusing is the result. Integral imaging in this thesis is mainly concerned with close imaging and parts of the scene are generally within *2f,* therefore, it could be argued that by moving the capture plane further away from the lens to the conjugate focus would improve the image. However, in integral photography, it is observed that when the capture plane is set at the focal plane of the lens array, parts of the scene closest to the array replay with the greatest focus. To explain this anomaly it is necessary to recognize that on capture, points closer to the array are imaged in fewer lenslets i.e. fewer lenslets 'see' these object points.

Subsequently, on replay, there are fewer lenslets involved in reconstructing these points, hence less room for error. Also another effect is present during decoding which gives greater clarity to points closer to the array, or more succinctly, less clarity to points further from the array. This is due to the considerable divergence of a pixel beam as it is projected from the lenslet. The further away from the lenslet that it travels the greater is the beam width; hence the more difficult it is to reassemble deeper points with clarity. This is associated with the voxel image of the point being large caused by many beams being reintegrated together to reform it. Moving the capture plane to a 'circle of least confusion' can only be performed for a given point and scenes contain thousands of points at different depths. Therefore, the capture plane positioned at the paraxial focal plane is a best compromise. Moving from this position exacerbates the situation for some areas of a scene and improves others.

Spherical aberration can be removed with two spherical lens arrays of opposite sign or minimized by adjustment of the shape factor of each lenslet in an array (ideal lens arrays) but more analysis is needed to ascertain any real benefit can be gained. Coma is associated with the shape of a point source originating from an off-axis position and this can be entirely eliminated by the same shape factor as that producing the minimum spherical aberration. Unfortunately aspheric surfaces for each lenslet in an array are very difficult to construct and such an aplanatic system would be very expensive. Other aberrations such as astimatism and curvature of field can be eliminated by the addition of a stop in front of each lenslet but this can lead to barrel distortion phenomena.

4.5 Integer and non-integer number of pixels per lens

A more fundamental problem than that of optical aberrations might be one associated with the distribution of pixels behind each lenslet. For example the 0.6mm pitch lens array at 300dpi has 7.077165 pixels behind each lenslet. If, for the sake of clarity, a lenslets first pixel is abutted against the interface between lenslets then the adjacent lenslet will have 0.077165mm of the eighth pixel coincident with its interface to the first lenslet. The next lenslet interface will be coincident with 0.154330mm into the fifteenth pixel (Figure 4.9). This progressive increase is perpetuated throughout the array and at every thirteenth lenslet

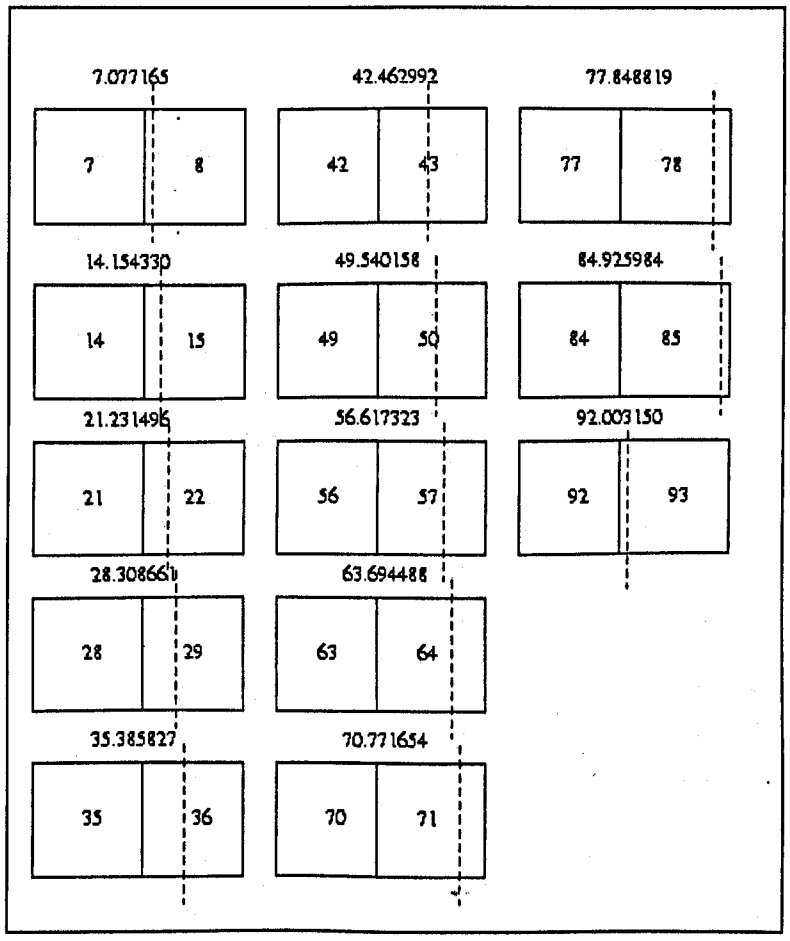

Figure 4.9: Traversing of the lens edge across each shared pixel

(pixel 93) the progression is greater than the pixel pitch and creates a jump from every seventh pixel to an eight pixel. Over a semi-cylindrical array of say 200 lenslets this amounts to a 'pixel drift' to the left and to the right of the centre pixel of 100/13 i.e. 7.692308 pixels out of phase or the equivalent of just over one lenslet in each direction. This effect can be seen from an integral sentence image stretching across the semi-cylindrical array (Figure 4.10). The further out from the centre of the array the more the letters lean over away from the centre.

To examine this effect more closely the finite-sized aperture model was configured to print out the spherically aberrated pixel hits from a single point using both a lens array with a non-integer number of pixels behind each lenslet (0.6mm pitch) and an array that has an integer number (1.27mm pitch), when both are generated at 300dpi. In order to easily make the comparison between the two the central lenslets mid-pixel is

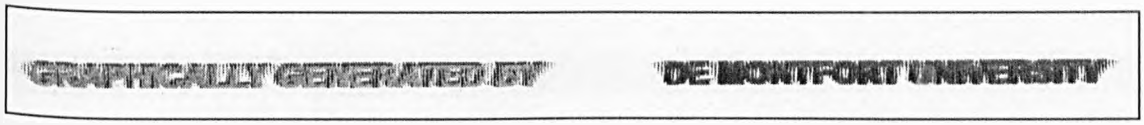

Figure 4.10: Integral image showing the leaning of letters towards each end due to pixel drift

numbered zero and a point is separately positioned, for each array, at the same distance and angle for each incrementing lenslet. Furthermore, the distances of the pixel hits, horizontally from each lenslet vertex, is counted as if the pixel level with the vertices was zero for all lenslets (see Appendix F).

Lenslet Pitch = 0.6mm

rays/lens = 30 no. of lenslets = 30 f =2.65 n_2 = 1.52 Xo=1.00 angle=0

vertex level pixel	pixel hits	lens vertex position
0	3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3	0.0
7	3 3 3 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3	0.6
14	3 3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2	1.2
21	3 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2	1.8
28	3 3 3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 -1 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2	2.4
35	3 3 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2	3.0
42	3 3 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2	3.6
50	2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3 -3	4.2
57	2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3 -3 -3	4.8
64	2 2 2 2 2 2 1 1 1 1 1 1 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3 -3 -3	5.4
71	2 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3 -3	6.0
78	3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3 -3	6.6
85	3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3	7.2
92	3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3	7.8
99	3 3 3 2 2 2 2 2 2 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -2	8.4
106	3 3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2	9.0
113	3 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2	9.6
120	3 3 3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 -1 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2	10.2
127	3 3 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2	10.8
134	3 3 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -1 -2 -2 -2 -2	11.4
142	2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3 -3 -3	12.0
149	2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3 -3 -3	12.6
156	2 2 2 2 2 2 1 1 1 1 1 1 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3 -3 -3	13.2
163	2 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3 -3	13.8
170	3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3 -3	14.4
177	3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3	15.0
184	3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3	15.6
191	3 3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -2	16.2
198	3 3 3 2 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2	16.8
205	3 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2	17.4

Lenslet Pitch = 1.27mm

rays/lens = 30 no. of lenslets = 30 f=3.08 n_2 = 1.52 Xo=1.00 angle=0

vertex_level_pixel	pixel hits	lens_vertex_position
0	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	0.00
15	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	1.27
30	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	2.54
45	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	3.81
60	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	5.08
75	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	6.35
90	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	7.62
105	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	8.89
120	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	10.16
135	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	11.43
150	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	12.70
165	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	13.97
180	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	15.24
195	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	16.51
210	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	17.78
225	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	19.05
240	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	20.32
255	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	21.59
270	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	22.86
285	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	24.13
300	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	25.40
315	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	26.67
330	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	27.94
345	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	29.21
360	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	30.48
375	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	31.75
390	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	33.02
405	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	34.29
420	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	35.56
435	4 4 4 4 4 4 4 4 3 3 3 2 2 1 1 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -4 -4 -4	36.83

The first set of pixel hits for the non-integer number of pixels behind each lenslet that is present in the 0.6mm pitch lens array reveals a repeating pattern. The whole block of a sequence can be seen from the actual vertex level pixel numbers 50 to 134, this is the thirteen lenslets before a jump of eight pixels to number 142. Whereas the 1.27mm pitch lens array reveals a totally homogeneous layout. The total pixel drift for any lens array size and pitch and any output resolution can be calculated by taking the decimal part of the number of pixels per lenslet and multiplying it by 25.4/dpi and this result is then multiplied by the number of lenslets.

From these results it seems clear that an integer number of pixels behind each lenslet is the most satisfactory but a non-integer number is usable. In fact an observer has to look hard to see any difference in quality between the two, with the exception, of course, that one has almost half the lenslet pitch than the other. Ideally the comparison should be between arrays closer in pitch, say, 0.6mm pitch and a pitch such as 0.5927mm that covers an integer number of pixels, therefore the results are indicative. Looking at the models discussed in Chapter 2, only the lens array model can properly use a lens array not covering an integer number of pixels whilst the 3D-from-2D model must use an integer number. This is another reason to use the lens array model for situations where the lens array and display type do not have an integer number. A further problem is that with a non-integer number of pixels per lenslet system a hit can be received on a shared pixel through one lenslet but this pixel intensity will replay back through the sharing lenslet to some other completely different point in object/image space. This effect has not been noticed in a replayed integral image. Considering Figure 4.9 the explanation may be that only a very small percentage is showing behind the first lenslet and the chances of being hit are equivalent to that percentage. This percentage does increase up to 50% at pixel number 43 but then diminishes again. So there is a major risk of this happening for only approximately 6 pixels out of 94.

4.6 Conclusions

The creation of a forward projection finite-sized aperture model that generates mesh integral images is described. The LeSDs generated from this model have spherical aberration and defocus effects as for images captured using real optics. This is due to each lenslet being modelled as a finite-sized aperture without stops. This chapter explains why spherical aberration and defocus do not present an obstacle to the generation of acceptable integral images and how this type of model allows a 'non-integer number of pixels per lenslet' system to be successfully employed.

The obvious main drawback associated with a finite-sized aperture model, which can be seen even in the production of mesh integral images, is that it is considerably slower computationally than the pinhole model due to the extra programming loop,

and therefore, at present day processing speeds, a better candidate for real-time generation is the pinhole model. The drawback in using a pinhole rendering model (as seen in later chapters), though, without any post processing, is that an 'integer number of pixels per lenslet' system is required.

The previous chapter and the present chapter have shown that integral imaging is possible using forward projection models. Lenslets are described in programming models as pinholes or finite-sized apertures and both generate a volumetric image when decoded by a lens array with the same geometries as their particular virtual lens array model. Carrying out the minimalistic model analysis has provided a sound understanding on which to base a more sophisticated approach and move to rendering programs described in detail in the following chapters.

5

Forward Projection Finite-Sized Aperture Rendering Model

5.1 Introduction

It is not possible to use standard interpolative shading methods unless all object points that make up the perimeter of an object triangle are simultaneously present at the image plane. This is not the case with the mesh programs reported in the previous two chapters as each object point is separately processed with no relative triangle perimeter linkage at the image plane that states 'this point belongs to this triangle'. The projection systems of Chapter 2, however, do take account of this as each object triangle perimeter is processed as a separate unit when it is tested for visibility from any part of the aperture, translated to the image plane, rendered using interpolative shading, and is tested again at a pixel level using a depth comparison hidden surface algorithm (i.e. z-buffer).

The model described in this chapter (see Fig 5.1) requires the drawing and saving of perimeter points in object space before translating the whole perimeter. In addition it

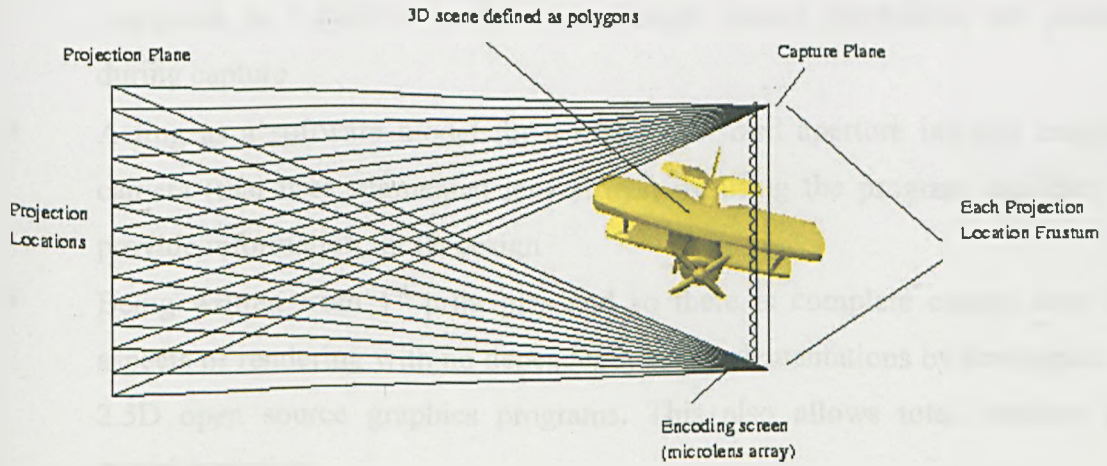

Figure 5.1: Finite-sized aperture rendering model

is necessary to store an identification marker for each image plane hit to enable the integrally modified standard shading algorithms to identify which side of the triangle the translated object point belongs to. Further, the intersection points of rays with the lenslets are now at curved surfaces and great accuracy is required to pinpoint these positions for the refraction calculations. The finite-sized aperture mesh model of Chapter 4 is *given* the intersection locations on the lenslets' curved surfaces but now the projector from projection point situated on the aperture plane through the object point is propagated to unknown intersections of these surfaces. Therefore, it is necessary to design the finite-sized aperture mesh model i.e. lens array model, from 1st principles.

It is clear that to produce integral images using a finite-sized aperture rendering model would be much slower than that of a pinhole model, but for quick (though not real-time) off-line graphics generation it has the advantages of:

- A non-integer number of pixels per lenslet can be used that enables any mix-and-match lens array and display system to be inexpensively employed together
- Overall perfect focus retains the depth of field but the non-uniform focus associated with finite-sized apertures gives an extra cue for depth perception
- No strobing effect during animation that is generally created when viewing a sequence of perfectly sharp images
- No major observed degradation of the integrity of the integral image when compared to a pinhole model even though optical aberrations are present during capture
- Acting as a software model for a real finite-sized aperture integral imaging camera (that uses segmented lens arrays) by using the program variables to provide information for its design
- Being written from 1st principles and so there is complete control over all aspects of rendering with no dependency on implementations by developers of 2.5D open source graphics programs. This also allows total freedom for experimentation.

5.2 Object file format

The object file format must necessarily contain the optical variables of the lens array besides the standard variables and modes found in modern or commercial 2.5D formats. To this end an integral imaging object file format is developed and evolved from 1st principles. The evolution begins with basic 3D coordinate object data with the variables manually input into the program and ends with a user-friendly format that contains all possible variables to be input at the beginning of each object file. This format contains all optical variables, colour information, shading modes, projection modes, dpi, and a range of techniques to allow full control over transforming chosen objects in a scene. The idea is to give as full a control as possible to the user for either static or dynamic displays and to allow this control to be easily understood and implemented.

The evolution of the format is not described here as the stage it has reached at present contains all previous stages; hence it is only necessary to describe the present stage. A two-step software process has also been developed to parse and translate VRML2 file formats containing H-anim 'Protos' to the integral imaging format scene file and to accept texture-mapping details. During the development stage modifications were made to the experimental programs to include the texture mapping of avatars using linear interpolations.

The variables in the integral imaging object file format are discussed in this chapter as they arise within their relevant functions within the integral imaging rendering software but below is an example of the initial set up details where for example:

Switches are ON for (global) INITIAL_SCALE: and (global) ROTATION: the latter denotes anti-clockwise rotation of 10° around the y-axis.

```
ANIMATION:          100
SHADING:            GOURAUD
ARRAY:              LENTICULAR
PROJECTION:         PERSPECTIVE
WEIGHTINGS:         0.8 0.5 0.6 40.0 0.48
BACKGROUND:         0 100 120
SOURCE:             0.5 0.5 0.5
LENSES:             2.116667 1.56 3.43718 1.0
APERTURE:           337.92
PROJ_LOCATIONS:     15
ARRAY_DEPTH:        0.5
SHIFTS:             1200 4364 201
OUTPUT_RES:         192.424242
LINE_DENSITY:       1.0
INITIAL_SCALE:      1 1.4 1.4 1.4
ROTATION:           1 10.0 0 1 0
TRANSLATION:        0 0.0 0.0 0.0
SCALING:            0 0.0 0.0 0.0
```

5.3 Implementation modes

Modified standard shading algorithms to colour the integral scene are used and these include flat, Gouraud and Phong shading techniques with the Phong reflection model. The experimental integral rendering programs were developed for semi-cylindrical lens arrays (lenticular) and spherical lens arrays (microlens) hence giving uni-directional and omni-directional parallax. Modifying the original experimental programs can make use of hexagonal lens arrays. Orthographic or perspective projections are also possible choices, perspective projection being a natural outcome when using spherical lens arrays. Altogether, this enables the implementation of any of nine possible modes (Table 5.1).

Mode	Shading	Lens Array	Projection
1	flat	Lenticular	Orthogonal
2	flat	Lenticular	Perspective
3	flat	Microlens	Perspective
4	Gouraud	Lenticular	Orthogonal
5	Gouraud	Lenticular	Perspective
6	Gouraud	Microlens	Perspective
7	Phong	Lenticular	Orthogonal
8	Phong	Lenticular	Perspective
9	Phong	Microlens	Perspective

Table 5.1: Nine possible implementation modes

5.4 Parameters

To enable the correct positioning of the virtual lens array within the scene, that is the scene portions to be replayed in front of and behind the real lens array, account has to be taken not only of the original scene depth but also the scene depth after the *initial scale* which sets up the required size of scene for the display. Therefore it is necessary to read in the scene Cartesian co-ordinates and scale them using the *initial scale* values entered in the scene file. The global depth is found after scaling and then the virtual lens array is positioned. To save processing time, and further file pre-reads, object centre coordinates (i.e. centres of separate objects) and local centre coordinates (separate colour-block parts of object centres) are found at this time and each object and local object is given an identification tag. This allows control over the transformation of individual objects or parts of objects within the scene. The C code to find the temporary position of the lens array and reposition the global centre z-axis position to world coordinates is:

```
zv_temp=(zv_pos*(gmax_z-gmin_z))+gmin_z;
global_centre.z=global_centre.z-zv_temp;
```

This pre-read of the scene file is necessary only once during capturing a sequence of frames, but for a single frame a dummy run is required; this is explained in section 5.6. For the first frame the user arbitrarily positions the virtual lens array and during animation the scene is allowed to develop its own natural positional relationship with the array. During the parsing of the scene file, to save processing time later, all parameters except for transformation and colour parameters are read and stored. Defaults are used for any parameters not entered into the scene file.

The final positioning of the aperture (i.e. projection point plane) on the optical axis (z-axis) to enable the start of the scene translation is performed by first calculating its required distance from the virtual lens array (i.e. d). For the purposes of gaining orientation control and enabling comparisons between different scenes for scaling and positioning on the relevant display the virtual lens array has its final position on the optical axis (zv) set at zero. The aperture plane is, therefore, the negative value of d.

Later, all z values (scene coordinates and virtual camera system) are shifted to accommodate for the array being positioned at zero on the optical axis.

The radius of curvature of the lenslets is required to be calculated as this parameter is used considerably in the translation of each object point through the lens array. It is at this point that it is relevant to complete a full optical matrix examination of a given lens array to ascertain its system matrix and provide the positions of the unit and focal planes and hence arrive at a clear understanding of the characteristics of the optics.

5.5 A lens array system matrix

One lenslet in an array of lenses is one of the simplest optical systems, a single lens, but lenslets in an array are plano-convex where the back of the lens (plano) is ideally coincident with the rear focal plane. In the plano-convex case the system matrix is simply $\mathbf{S} = \mathbf{T_1}\mathbf{R_1}$ where $\mathbf{T_1}$ is the translation matrix and $\mathbf{R_1}$ is the refraction matrix [56]. There is no refraction matrix for the back of the lens as it is a plane surface and as such produces a unit matrix and hence there is no effect when multiplying by other matrices. The elements of the system matrix are the Gaussian constants of the optical system and are depicted as *a, b, c*, and *d* from which the unit planes, focal planes and focal lengths can be calculated i.e. F, H, F^1, H^1, f, f^1 respectively (Figure 5.2).

The system matrix is derived thus:

$$\mathbf{T_1} = \begin{bmatrix} 1 & 0 \\ t^1/n^1 & 1 \end{bmatrix} \qquad \mathbf{R_1} = \begin{bmatrix} 1 & -k \\ 0 & 1 \end{bmatrix}$$

Where t^1 is the thickness of the lenslet, n^1 is the refractive index of the lenslet and

$$k = \frac{n^1 - n}{r} = \frac{1}{f}$$

In this equation n is the refractive index of air and taken to be unity, thus:

$$\mathbf{S} = \begin{bmatrix} b & -a \\ -d & c \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ t^1/n^1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -k \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -k \\ t^1/n^1 & 1 - kt^1/n \end{bmatrix}$$

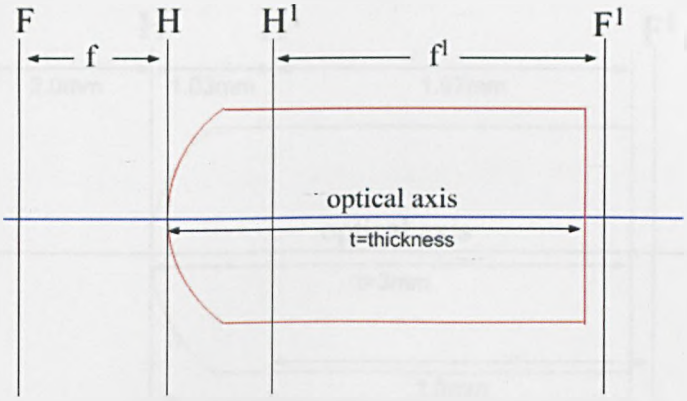

Figure 5.2: Cardinal points of a lenslet in a lens array

where from the system matrix:

$$a = k \qquad b = 1 \qquad c = \frac{1 - kt^1}{n^1} \qquad d = \frac{t^1}{n^1}$$

The planes and lengths can be derived from the Gaussian constants as:

Unit Planes

$$\text{Front (H)} = \frac{1-b}{a} \qquad \text{Rear (H}^1\text{)} = \frac{c-1}{a}$$

Focal Planes

$$\text{Front (F)} = \frac{-b}{a} \qquad \text{Rear(F}^1\text{)} = \frac{c}{a}$$

Focal Lengths

$$\text{Front (f)} = \frac{1}{-a} \qquad \text{Rear (f}^1\text{)} = \frac{1}{a}$$

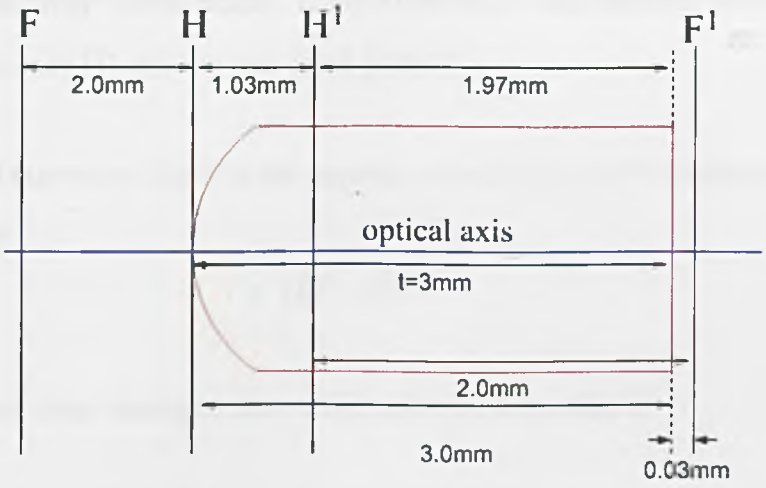

Figure 5.3: Annotated cardinal points of a lenslet in a lens array

For a lens array with lenslet pitch of 1.27mm, refractive index of 1.52, a focal length of 2mm, and thickness of 3mm results in:

$$\mathbf{S} = \begin{bmatrix} b & -a \\ -d & c \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1.97 & 1 \end{bmatrix} \begin{bmatrix} 1 & -0.5 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -0.5 \\ 1.97 & 0.015 \end{bmatrix}$$

From the final matrix it can be seen that the Gaussian constants have the values:

$$a=0.5 \quad b=1 \quad c=0.015 \quad d=-1.97$$

therefore:

$$H=0 \quad H^1=-1.97 \quad F=-2 \quad F^1 \quad f=-2.0 \quad f^1=2.0$$

Negative values move from right to left and Figure 5.3 is annotated to show all values. Both H, H^1 and F, F^1 are measured from their respective surfaces and the focal lengths are between H/F and H^1/F^1. It can be seen from these calculations that using the rear focal length to position the final hit during virtual capture at the back surface of each lenslet would be incorrect. However, the position of H^1 is not coincident with H and the lenslet vertex, therefore the value to be used, not only during the projection of the rays but also for the important calculation of *d* (distance between projection point

plane and lens array vertex plane), is $t + 0.03(mm)$ - the distance between the lenslet vertex (in this case H) and the rear focal plane F^1.

The radius of curvature, used in the capture calculations, can be derived from the thin lens equation:

$$r = f(n^1 - n)$$

and, in the lens array example above, the radius of curvature is:

$$r = 2.0(1.52 - 1) = 1.04mm$$

which, in this case, is almost coincident with the rear unit plane H^1.

5.6 Initial scene set-up and animation

The procedure **Dummy Run** runs through the scene file and using *all* its transformations ascertains the final scene depth to enable the required positioning of the lens array. It is used when the first frame of a sequence of an animation (set ANIMATION: 1) is to be set up or in the production of a single integral image.

The user might need to re-position and view the objects within the scene before the actual production of a sequence of images. Or, the user might only require a single static integral image but still needs to set it up according to object/colour-block positional requirements. In the procedure the transformation data of the scene is read in and objects within the scene and colour-blocks are translated, rotated and/or scaled to find their required positions and orientations in the scene. To perform the rotation on an individual aspect of the scene requires the use of all the rotational centres previously calculated in **Parameters** (section 5.4). The value of the temporary position of the lens array *(zv_temp),* based purely on a global transformation, can now be re-calculated.

For ANIMATION>1 the first frame is only allowed to be transformed by the *initial scale* and then the next sequence of frames alters the object data for each frame by

using the scene file transformation details already read in and stored. There is no need to use a 'dummy run'.

An animation run loops through the tracing and rendering procedures for the same number of times as that set in the scene file by the user or just once for a single frame. In this loop the scene data passes through a general cull (section 4.2), the triangle perimeters are calculated and stored (section 3.3), normals and vertex normals of the triangles are found (sub-section 5.7.2), object points are transferred to the image plane (sub-section 5.7.4) and modified shading algorithms are engineered. Throughout these processes there are many specific items requiring novel attention in order for the final image to be in integral format.

5.7 Trace and shade

The techniques used in this section are explained by referring to the treatment of one triangle projected from a single position on the projection point plane. The process is repeated for every position and for each triangle in turn, whereby the captured intensity distribution is built up in the virtual recording medium. Each cycle of the loop is counted by the variable t, beginning at $t=0$, and this is used to increment the rotation, translation and scaling values if any of these switches within the scene file are ON.

5.7.1 Object and Colour-Block Transformations

Each object in the scene file has an associated whole number in order for the transformations that are switched ON for that particular object to be utilised within the program. Colour-blocks are separately identified using a similar identification strategy. Here is an example of this part of the scene file:

The switches are ON for ROTATE_OB: TRANSLATE_OB: SCALE_OB for all colour-blocks in OBJECT 1, and ROTATE: as an extra transformation in the first colour-block. As this part of the scene file follows directly on from the example in

section 5.2 it can be seen that the whole scene is also transformed by INITIAL_SCALE: and ROTATION: .

```
SCENE
OBJECT              1
ROTATE_OB:          1 115.0 0 0 0
TRANSLATE_OB:       1 0.0 -40.0 0.0
SCALE_OB:           1 2.15 2.15 2.15
COLOUR 1            25 25 25
ROTATE:             1 20.0 0 0 0
TRANSLATE:          0 0.0 0.0 0.0
SCALE:              0 0.0 0.0 0.0
TRIANGLES:
-24.871553 16.515732 -115.871414 -20.012758 27.904732 -102.460320 -
16.029018 13.646645 -91.795341
-24.871553 16.515732 -115.871414 -16.029018 13.646645 -91.795341 -
20.887846 2.257645 -105.206406
-24.871553 16.515732 -115.871414 -22.704788 16.527142 -116.665504 -
17.846127 27.915859 -103.254753
COLOUR 2        250 250 250
ROTATE:          0 20.0 0 0 0
TRANSLATE:       0 0.0 0.0 0.0
SCALE:           0 0.0 0.0 0.0
TRIANGLES:
-20.873987 2.205559 -105.167168 -16.015202 13.594551 -91.756081 -
12.031418 -0.663658 -81.091003
-20.873987 2.205559 -105.167168 -12.031418 -0.663658 -81.091003 -
16.890251 -12.052666 -94.502083
```

If ANIMATION: is set to 1 i.e. a single frame, and either one or both the ROTATE_OB or ROTATE: switches are ON then the angle of rotation is immediately used in the program. Similarly, the object and colour-block translations and scaling are directly implemented. However, if the animation setting is greater than 1, any transformation data is multiplied by t, and as the first value of t is zero there will be no transformations for the first frame. Each successive frame, though, is transformed by the multiplication of an incrementing integer.

It is during the calculations of the transformations that all scene z-coordinates are brought into conformity with the lens array, positioned at zero on the optical axis, by subtracting from them the value of *zv_temp* (section 5.4) i.e. *z1=z1-zv_temp.* As the actual transformation calculations are used by standard computer graphics it will not be dealt with here except to say that the global centre, object centre and local centre

xyz coordinates must be subtracted from the scene coordinates before rotation and added back on at the end of these calculations for the rotations to perform as required.

In using this program, during a 'sequence of frames' run, the facility of saving to file (in the integral imaging scene file format) the last positions of all items in the scene has been devised. This allows the user to change transformation directions and continue producing more sequences, moving chosen objects in chosen ways, for the desired final animation. The file is built and saved when t is the value of the last frame count minus 1 (as t began the cycle at a value of zero). Again, for conformity, the z-coordinate values of the scene data are returned to their object file state by adding back *zv_temp*.

5.7.2 Normals

The general cull that strips off triangles from the scene database that cannot be seen from any part of the aperture was explained in section 4.2 and is the next stage of the program. Immediately following this stage the triangle normals are calculated for modes in which flat shading is required and these are stored in an array that reflects the array containing the scene database. The normal of each triangle is calculated from the cross product of two of the triangle vectors.

For the purposes of Gouraud and Phong shading, the vertex normals must also be calculated and stored in the same way as the triangle normals. Each triangle corner is shared with connecting triangles, which is the basis for calculating the vertex normals, and in this program it is only necessary to calculate these once. This is implemented by checking all the triangle vertices against each other. Those that are the same are given the same number and the normals of the triangles (of which the vertices are a member) are averaged. These are the vertex normals. Each vertex normal is calculated once, and only the vertices yet without a number are considered during the running of the algorithm. The process is efficient and moves progressively faster.

These values, like the triangle normals, can also be pre-processed and entered as part of the object database, where on input they are transformed along with the objects to which they belong. The calculation of normals and vertex normals are standard

computer graphics techniques but the algorithm providing the vertex normals is an original contribution and is both complex and fast due to the coding technique and algorithm structure. It needs only to be performed once before tracing and rendering and is listed following this text.

```
void vector_norm()
{
 int i,j,k,p=1,flag=0,vert_num[VERTMAX];

 for (i=0;i<vertsize;i++)   vert_num[i]=-1.0;

 for (i=0;i<vertsize;i++)
  {
   if (vert_num[i]>-1.0)  flag++;
     else
       {
        vect_copy(&vect_norm[i],&norm[i]);
        vert_num[i]=i-flag;
        for(j=i+1;j<vertsize;j++)
          {
           if (j==vertsize-1)
              {
               if
                (
                (v[j].x==v[i].x)&&(v[j].y==v[i].y)&&(v[j].z==v[i].z))
                 {
                  p++;
                  vert_num[j]=i-flag;
                  vect_add(&vect_norm[i],&vect_norm[i],&norm[j]);

                 }
                  vect_norm[i].x=vect_norm[i].x/p;
                  vect_norm[i].y=vect_norm[i].y/p;
                  vect_norm[i].z=vect_norm[i].z/p;
                  for (k=i+1;k<vertsize;k++)
                    if (vert_num[k]==vert_num[i])
                      {
                        vect_copy(&vect_norm[k],&vect_norm[i]);
                        p=1;
                      }
               }
                if
((v[j].x==v[i].x)&&(v[j].y==v[i].y)&&(v[j].z==v[i].z)&&(j!=vertsize-
1))
                  {
                   p++;
                   vert_num[j]=i-flag;
                   vect_add(&vect_norm[i],&vect_norm[i],&norm[j]);
                  }
          }
      }
  }

}
```

In the algorithm the number of scene vertices is given the variable *vertsize*; *vertnum[]* is the array containing the number given to each vertex; *vect_norm* is self explanatory; *v[].x, v[].y, v[].z* contain the scene coordinates.

5.7.3 Ambient and diffuse components

Before moving through the resultant scene database one triangle at a time and project them through the lens array for each projection location the drawing of each perimeter in 3D object space (sub-section 3.3.2) is performed and the number of coordinates that represent those perimeters (**perim**) is returned. A single light source vector is set up in *lvec[].x, lvec[].y* and *lvec[].z*. The ambient and diffuse attributes of the corners of the triangles are independent of the 'projection point through object point' projector direction and as such, for flat and Gouraud shading, can be evaluated before multiple passages through the lens array with respect to each projection point, hence saving unnecessary repeated calculations. The specular contribution, however, must be calculated within the changing loop that steps along the projection points as it is determined by the angle produced between the projector and the mirrored light vectors.

The ambient and diffuse attribute calculations are different for flat shading and Gouraud shading and as such are called upon by looking for the particular mode that is set in the scene file i.e. SHADING: ARRAY: PROJECTION:. The values entered by the user in WEIGHTINGS: are the Phong reflection model variables (Chapter 1). These are, in turn, the ambient constant, the wavelength-dependent empirical reflection coefficient related to the diffuse light, the specular contribution as a function of the angle between the viewing direction and the mirror direction, followed by a value that is an index that simulates surface roughness and finally a brightness intensity control. These can be set up and experimented with as suits the user.

5.7.4 Tracing rays from aperture to image plane

The lens arrays provided for in the program are both semi-cylindrical and spherical microlens arrays, the former giving parallax in the horizontal direction, the other omni-directional parallax. Obviously the latter, the spherical microlens array, requires

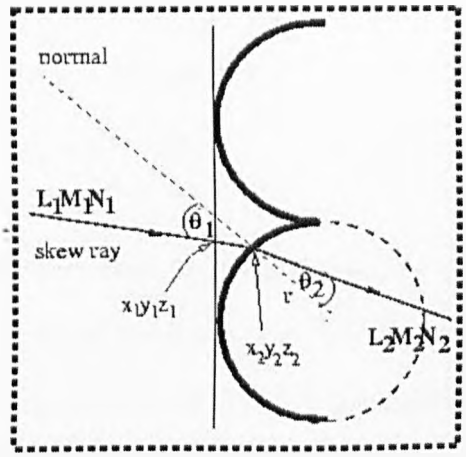

Fig. 5.4: Transfer and refractive process

more information from the scene. For omni-directional parallax the aperture or projection point plane is modelled as a square with the projection locations distributed over the square in equally spaced rows and columns. The semi-cylindrical array requires an infinitely thin aperture with the projection locations distributed at equal spacings, hence, considerably less projection locations are required.

In introduction to this chapter it is explained that projectors traverse from projection point locations through the object points and they can hit any lenslet in a lens array and anywhere on its curved surface (Figure 5.4). The exact coordinate information of each hit is required to enable the correct refraction calculations to be made and hence enable the determination of the final resting place of each ray at the image plane. To accommodate this, the universally used system for tracing skew rays through spherical surfaces is used [57]. This transfer and refractive process uses the standard properties of direction cosines, the refractive index and surface of curvature (c) (Eqn.5.1) of the lenslets. A quadratic equation can be formed (Eqn.5.2) to describe the transfer process and the solution is written in (Eqn.5.3).

$$z = \frac{c(x_2^2 + y_2^2 + z_2^2)}{2} \qquad \text{(Eqn.5.1)}$$

$$D^2 - 2FrD + Gr = 0 \qquad \text{(Eqn.5.2)}$$

$$D = r\left(F - \sqrt{F^2 - G/r^2}\right) \qquad \text{(Eqn.5.3)}$$

Where:

$$F = N_1 - c(L_1 x_1 + M_1 y_1 + N_1 z_1)$$
$$G = c(x_1^2 + y_1^2 + z_1^2) - 2z_1$$

r is the radius of lenslet curvature, and (x_1, y_1, z_1) are the coordinates of the ray intersection at the (x, y) plane of the lenslets apex. D is the distance from (x_1, y_1, z_1) to the lenslet surface (x_2, y_2, z_2) Using D and the properties of direction cosines the ray intersect with the lenslet surface can be evaluated. Further, the refractive formulae can be derived (Eqns.5.4).

$$\left.\begin{aligned} \cos\theta_1 &= N_1 - c(L_1 x_2 + M_1 y_2 + N_1 z_2) \\ n_2 \cos\theta_2 &= \sqrt{n_2^2 - 1 + \cos^2\theta_1} \\ \sigma &= c(n_2 \cos\theta_2 - \cos\theta_1) \end{aligned}\right\} \text{(Eqns.5.4)}$$

Consequently, the new refracted direction cosines can be found from Eqns.5.5 where θ_1 is the angle of incidence, θ_2 is the angle of refraction, n_2 is the refractive index of the lenslets.

$$\left.\begin{aligned} n_2 L_2 &= L_1 - \sigma x_2 \\ n_2 M_2 &= M_1 - \sigma y_2 \\ n_2 N_2 &= N_1 - \sigma(z_2 - r) \end{aligned}\right\} \text{(Eqns.5.5)}$$

The new values are used to find the final coordinates at the image plane. Of course, calculations are also made to determine which lenslet of the array is hit.

To implement these equations in the programming code it is necessary to find the direction cosines of the ray (*ll, mm, nn*) that is incident at the object point from a radiant on the aperture. Simply, a direction cosine is the 2D distance the line moves from the starting point coordinates relative to those individual coordinates divided by the actual 3D length of the ray. The distance the line moves relative to each coordinate is given the variable *incident_vec* and the object point is stored in an array

called *lines[]* and *x_pt*, *y_pt* and *z_pt* are the variables that denote the starting coordinates on the aperture plane. These are implemented differently for the three groups of modes i.e. lenticular orthoscopic, lenticular perspective and microlens (meaning spherical lens). The lenticular orthoscopic modes are modes that 'stamp' the scene onto the image plane (via the lens array) and as such do not have a variance between starting point and ending point in the x-direction i.e. same direction as the semi-cylindrical lenslets. The lenticular perspective modes and the microlens modes produce skew rays and as such the calculations of the initial direction cosines are the same. These are all shown in C code:

Lenticular Orthoscopic

```
for (k=0;k<perim;k++)
      {
        incident_vec.x=0;
        incident_vec.y=lines[k].y-y_pt;
        incident_vec.z=lines[k].z-z_pt;
        length=vect_mag(&incident_vec);
        ll=0;
        mm=incident_vec.y/length;
        nn=incident_vec.z/length;
```

Lenticular Perspective and Microlens

```
for (k=0;k<perim;k++)
      {
        incident_vec.x=lines[k].x-x_pt;
        incident_vec.y=lines[k].y-y_pt;
        incident_vec.z=lines[k].z-z_pt;
        length=vect_mag(&incident_vec);
        ll=incident_vec.x/length;
        mm=incident_vec.y/length;
        nn=incident_vec.z/length;
```

vect_mag returns the square root of the added *incident_vec* squares.

The rays must next be projected until they reach the lens array vertex plane (*reach_point*) that has been set at zero on the optical axis. This is a simple matter when using the laws of direction cosines:

Lenticular Orthoscopic

```
reach_point.x=lines[k].x;
reach_point.y=lines[k].y+((mm/nn)*(zv-lines[k].z));
reach_point.z=zv;
```

Lenticular Perspective and Microlens

```
reach_point.x=lines[k].x+((ll/nn)*(zv-lines[k].z));
reach_point.y=lines[k].y+((mm/nn)*(zv-lines[k].z));
reach_point.z=zv;
```

Note that in the case of the orthoscopic mode *reach_point.x* is the same value as the objects points x-coordinate – this is the 'stamping' action.

The main difficulty in tracing the rays through the lens array is to find the 3D length of the short line from $x_1y_1z_1$ to $x_2y_2z_2$ (D) – see Fig. 5.4. This depends on the incident position (and angle) of the intersection of the lens array vertex plane by the ray, but which lenslet is the ray to be refracted by, and where the intersection is with respect to the curved surface of that lenslet need to be found. The transfer and refractive equations can be used to do this. The equations were originally derived for a curved surface at the optical axis, but, of course, in an array of lenslets there are as many optical axes as there are lenslets. The optical axis is, however, generally known as a line with zero values of x and y and the vertex of a curved surface has a zero z-coordinate. The lens array vertex (*zv*) in the program *has* already been set at zero. To enable the use of the equations the vertex of each lenslet, when 'chosen' by a ray, is mathematically brought to this position and, after the calculations, the distance along the x and y-axis that it is moved is added on to the end-point of the refracted ray. When using the semi-cylindrical modes this distance is only in the y-direction. The vertex of the lenslet under consideration (*xv,yv* or just *yv*) is calculated and this technique now gives a new temporary reach_point (*temp_reach_point*):

Lenticular Orthoscopic and Perspective

```
yv=(int)(floor((reach_point.y/pitch)+0.5)*pitch);
temp_reach_point_y=reach_point.y-yv;
```

Microlens

```
yv=(int)(floor((reach_point.y/pitch)+0.5)*pitch);
temp_reach_point_y=reach_point.y-yv;
xv=(int)(floor((reach_point.x/pitch)+0.5)*pitch);
temp_reach_point_x=reach_point.x-xv;
```

The transfer equations can now be employed where *fff*, *gg* and *delta* are the code representations of F, G and D in equations 5.2 and 5.3:

Lenticular Orthoscopic

```
fff=curv*(temp_reach_point_y*temp_reach_point_y);
gg=nn-(curv*(mm*temp_reach_point_y));
delta=fff/(gg+sqrt((gg*gg)-(curv*fff)));
chord.x=reach_point.x;
chord.y =temp_reach_point_y+(mm*delta);
chord.z =(nn*delta);
```

Lenticular Perspective

```
fff=curv*(temp_reach_point_y*temp_reach_point_y);
gg=nn-(curv*(mm*temp_reach_point_y));
delta=fff/(gg+sqrt((gg*gg)-(curv*fff)));
chord.x =reach_point_x+(ll*delta);
chord.y =temp_reach_point_y+(mm*delta);
chord.z =(nn*delta);
```

Microlens

```
fff=curv*((temp_reach_point_x*temp_reach_point_x)+(temp_reach_point_y
*temp_reach_point_y));
gg=nn-(curv*((ll*temp_reach_point_x)+(mm*temp_reach_point_y)));
delta=fff/(gg+sqrt((gg*gg)-(curv*fff)));
chord.x =temp_reach_point_x+(ll*delta);
chord.y =temp_reach_point_y+(mm*delta);
chord.z =(nn*delta);
```

where the *chord* variables are the coordinates on the lenslet surface ($x_2y_2z_2$ in Figure 5.4).

The refractive and new direction cosine equations can now be implemented:

Lenticular Orthoscopic

```
cos_thet1=nn-(curv*((mm*chord.y)+(nn*chord.z)));
cos_thet2=sqrt((n2*n2)-1+(cos_thet1*cos_thet1))/n2;
kk=curv*((n2*cos_thet2)-cos_thet1);
mm2=(mm-(kk*chord.y))/n2;
nn2=(nn-(kk*(chord.z-(1/curv))))/n2;
x_fin=chord.x;
y_fin=yv+chord.y+((mm2/nn2*(zv+thick-chord.z)));
```

Lenticular Perspective

```
cos_thet1=nn-(curv*((mm*chord.y)+(nn*chord.z)));
cos_thet2=sqrt((n2*n2)-1+(cos_thet1*cos_thet1))/n2;
kk=curv*((n2*cos_thet2)-cos_thet1);
mm2=(mm-(kk*chord.y))/n2;
nn2=(nn-(kk*(chord.z-(1/curv))))/n2;
x_fin=lines[k].x+((ll/nn)*(zv+thick-lines[k].z));
y_fin=yv+chord.y+((mm2/nn2*(zv+thick-chord.z)));
```

Microlens

```
cos_thet1=nn-(curv*((ll*chord.x)+(mm*chord.y)+(nn*chord.z)));
cos_thet2=sqrt((n2*n2)-1+(cos_thet1*cos_thet1))/n2;
kk=curv*((n2*cos_thet2)-cos_thet1);
ll2=(ll-(kk*chord.x))/n2;
mm2=(mm-(kk*chord.y))/n2;
nn2=(nn-(kk*(chord.z-(1/curv))))/n2;
x_fin=xv+chord.x+((ll2/nn2*(zv+thick-chord.z)));
y_fin=yv+chord.y+((mm2/nn2*(zv+thick-chord.z)));
```

Notice that the final position at the image plane is reached (*x_fin, y_fin*) by using the focal plane position of the lens array (*thick*) not the focal length (section 5.5) – the ideal being that the focal plane is on the back surface of the lens array. The distance of the lenslet (*xv,yv*) from the optical axis of the central lenslet is added back on to the position of the ray. Notice also that for the lenticular perspective mode the calculation for *x_fin* does not include a value of refraction but continues to use the original direction cosines. This is because the semi-cylindrical lenses are one dimension (x-dimension) short when compared to spherical lenses that refract in all directions. The

lenticular orthoscopic mode has a value for *x_fin* of *chord.x* but *chord.x* equals *reach_point.x* which equals *lines[].x*; it has only been left in the code for comparison with the other modes.

The final step to take in this series of equations is to correct the size of the captured perimeters in order to satisfy the resolution (dpi) requirements of the output display or hardcopy printout (section 3.5) and pixelate the image. This step is the same for all the modes:

```
xf=(int)floor((x_fin*sc)+0.5);
yf=(int)floor((y_fin*sc)+0.5);
xf=xf+vert_shift;
yf=yf+horiz_shift;
```

Where $sc = dpi/25.4$. The *vert_shift* and *horiz_shift* are variables input by the user in the scene file to position the integral image on the display.

5.7.5 Debugging

At this stage of the program it is possible to generate and produce hardcopy of images of the ray incidences with the lens array. This can be used as a quantitative method devised here to check that the rays are hitting the virtual lens array correctly. The first images revealed that something was not quite right with the code (Fig 5.5 i, ii). The first image (i) shows that many lens edge hits stream away from the correct virtual lenslet positions. The problem here is that the rays projected close to the boundaries between lenslets have their intersections calculated for an adjacent lenslet due to lens sag and ray direction and throw the lenslet intersection calculations awry. Another ray incident problem presented itself (ii) and it was necessary to isolate and enlarge a single lenslet to determine the solution, in this case a simple coding error. After modifying code was added along with code corrections, side view images of the hits on both virtual semi-cylindrical and spherical lens arrays (Fig 5.6 i, ii) were produced. The first image (i) correctly reveals the profile of the semi-cylindrical microlenses because the curvature is in one direction, whilst the spherical microlenses (ii) shows filled lenses due to all round curvature. Figure 5.7 is a front view of ray intersections on a virtual spherical lens array in which the object is a teapot and can be seen to

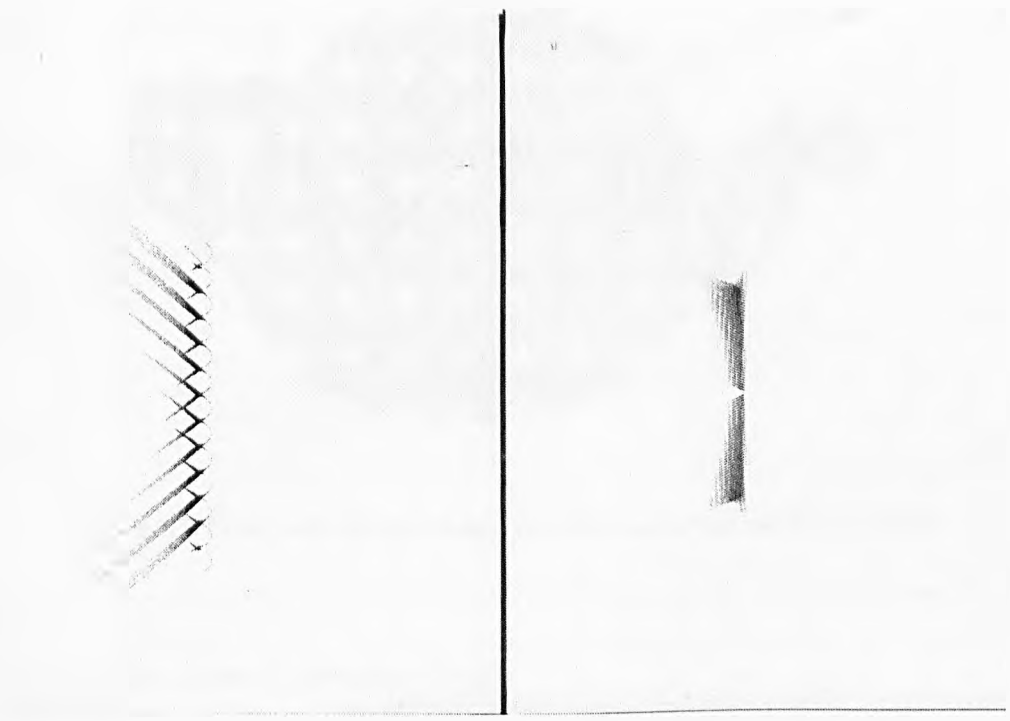

Figure 5.5: Incorrect ray incidences with the virtual lens array

Figure 5.6: Correct ray incidences with the virtual lens array i) side view of cylindrical lens array hits ii) side view of microlens array hits

generally confirm the correctness of the approach. Qualitative numerical checks can also be carried out. For the transfer process the coordinates of the point of incidence must satisfy the equation of the refracting surface (Eqn.5.1), and for refraction the sum of the squares of the direction cosines must equal unity i.e. $L^2 + M^2 + N^2 = 1$.

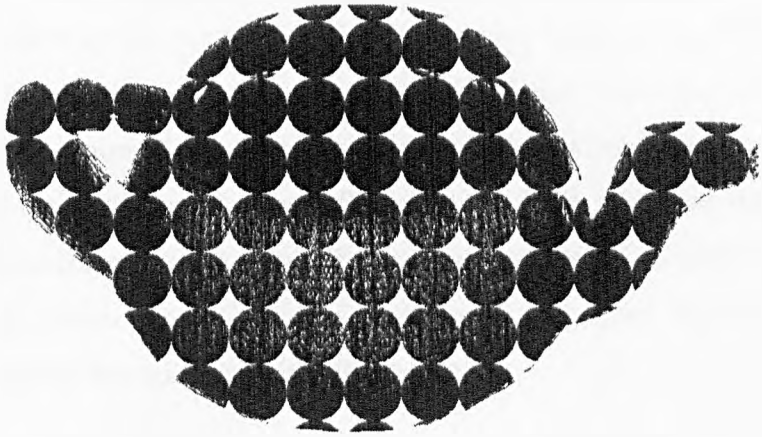

Figure 5.7: Front view of microlenses revealed by correct ray intersections

5.7.6 Group identities of perimeter lines

Standard computer graphics techniques for interpolative shading perform the interpolation of intensity values - and floating point depth values for the z-buffer – from assigned intensities of the three corners of the transferred triangle at the image plane. To do this it is necessary to simultaneously use a 2D line-drawing algorithm that has to check the orientation of the corners in order to know which corner is above, below, left or right from another corner to interpolate and rasterise. However, the finite-sized aperture integral imaging model transfers the whole perimeter through a lens array and the resultant effect at the image plane is not a set of linear sequences of pixels but discontinuous lines for each perimeter side. A further technique to find the orientation of each pixel used in making up a line is therefore necessary. If an identity for each final pixel value of each transferred object perimeter point can be maintained immediately after each transfer then orientation can be resolved and allow interpolative shading to proceed.

The first line of code in sub-section 5.7.4 (`for (k=0;k<perim;k++)`) loops triangle perimeter points through the above sequences until all perimeter points (*perim*) are transferred to the image plane. During the running of the 3D line-drawing algorithm, counts of all the points making up each of the three lines of a triangle are made and stored in *points[0]*, *points[1]* and *points[2]*. After each pixel coordinate for each

perimeter point is calculated a check can be made to see if the present value of *k* i.e. the present number of the perimeter point in question, belongs to *points[0]*, *points[1]* or *points[2]*. The triangle corners are numbered 1, 2 and 3 and the lines between the corners are given numbers relative to the corner numbers i.e. 12, 23, 31. This information along with the pixel coordinates is stored in the pre-buffer array and is essentially discontinuous edge lists. The y-pixel coordinate of each corner and the depth of each corner point are also stored for their part in the interpolation, rasterisation and hidden surface removal process.

Finally, it is convenient at this point to check each pixel coordinate hit to ascertain the extreme left, right, up and down values to enable the drawing of a bounding box around each triangle and save processing time later by only searching for hit pixels within that box. For purposes of debugging it has been possible to generate an integral image of omni-directional triangles showing the bounding boxes working (Figure 5.8). Initially, as seen in the image, the first bounding box incorrectly contains zero-coordinate information that increased processing time due to searching for edges over a much higher area than was necessary. After code correction the problem was solved and processing time saved. The code to perform all of the above storing, identification and the generation of the bounding box perimeters is:

```
if ((k>=0)&&(k<points[0]))              /* 1st side */
        {
         pre_buffer[xf][yf]=12;         /* identification */
         pixy[1]=yf;                    /* y-value stored */
         depth[1]=lines[k].z;           /* depth value stored */
        }
if ((k>=points[0])&&(k<(points[0]+points[1]))) /* 2nd side  */
{
         pre_buffer[xf][yf]=23;
         pixy[2]=yf;
         depth[2]=lines[k].z;
        }
if ((k>=(points[0]+points[1]))&&(k<(points[0]+points[1]+points[2])))
        {                               /* 3rd side */
         pre_buffer[xf][yf]=31;
         pixy[0]=yf;
         depth[0]=lines[k].z;
        }
if (yf>yup)     yup=yf;       /*generating the bounding box peimeter*/
if(yf<ydown)    ydown=yf;
if (xf>xright) xright=xf;
if (xf<xleft)  xleft=xf;
```

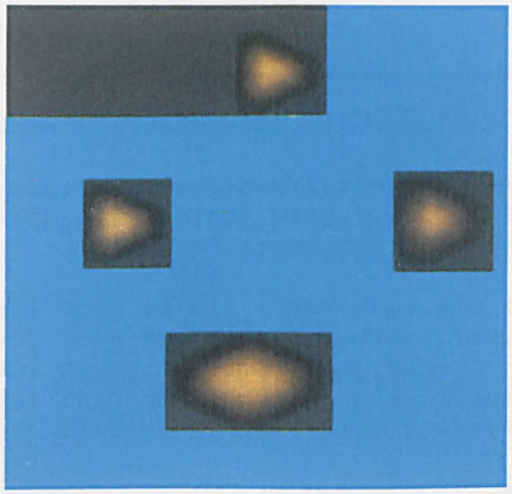

Figure 5.8: Integral image showing an incorrectly appointed bounding box

5.7.7 Interpolative shading

The specular highlight values for flat and Gouraud shading can now be calculated and added to the ambient and diffuse values already processed outside the perimeter point transfer loop. In the case of flat and Gouraud shading these intensities are only calculated for the three corners of each triangle and are then bi-linearly interpolated to fill the inner triangle pixels. However, Phong shading bi-linearly interpolates the vertex normals and calculates the ambient, diffuse and specular components of the intensity values for each relevant pixel and, as such, these cannot yet be computed. Each shading technique requires different attention and therefore different functions are called depending on the mode number (see Table 5.1). This is set up in code as:

Microlens

```
if ((mode==3)||(mode==6)||(mode==9))
   {
    for (y_pt=-pos_neg;y_pt<=pos_neg;y_pt=y_pt+otis_incre)
      for (x_pt=-pos_neg;x_pt<=pos_neg;x_pt=x_pt+otis_incre)
        if(reach_points_ulens(points,pixy,i,zv,&v1,&vvec,vvvec,
                              y_pt,x_pt,lines,perim,sc)==0)
          {
           if (mode==3) spec_am_dif_flat(&iadd,&io,&llvec,&vvec,i);
           if (mode==6) spec_am_dif_gouraud(iad,iog,lvec,vvvec,i);
           if (mode==3) fill_flat_lent(pixy,&io);
           if (mode==6) fill_gouraud_lent(pixy,iog);
           if (mode==9) fill_phong_lent(pixy,i,&Ns,&llvec,&vvec,&io);
          }
    }
```

Lenticular Perspective

```
if ((mode==2)||(mode==5)||(mode==8))
  {
    for (y_pt=-pos_neg;y_pt<=pos_neg;y_pt=y_pt+otis_incre)
      if (reach_points_pers(points,pixy,i,zv,&v1,&vvec,vvvec,
                            y_pt,lines,perim,sc)==0)
        {
         if (mode==2) spec_am_dif_flat(&iadd,&io,&llvec,&vvec,i);
         if (mode==5) spec_am_dif_gouraud(iad,iog,lvec,vvvec,i);
         if (mode==2) fill_flat_lent(pixy,&io);
         if (mode==5) fill_gouraud_lent(pixy,iog);
         if (mode==8) fill_phong_lent(pixy,i,&Ns,&llvec,&vvec,&io);
        }
  }
```

Lenticular Orthogonal

```
if ((mode==1)||(mode==4)||(mode==7))
  {
    for (y_pt=-pos_neg;y_pt<=pos_neg;y_pt=y_pt+otis_incre)
      if (reach_points_ortho(points,pixy,i,zv,&v1,vvvec,
                             y_pt,lines,perim,sc)==0)
        {
         if (mode==1) spec_am_dif_flat(&iadd,&io,&llvec,&vvec,i);
         if (mode==4) spec_am_dif_gouraud(iad,iog,lvec,vvvec,i);
         if (mode==1) fill_flat_lent(pixy,&io);
         if (mode==4) fill_gouraud_lent(pixy,iog);
         if (mode==7) fill_phong_lent(pixy,i,&Ns,&llvec,&vvec,&io);
         }
  }
```

The *spec_am_dif_gouraud* function calculates the pixel intensities of the corners of each triangle, *spec_am_dif_flat* calculates the intensity of the whole triangle. The *fill...* functions (except for *fill_flat_lent*) are the interpolating functions where *fill_phong_lent* contains within it the call of *spec_am_dif_phong* (see subsections 5.7.7.1 and 5.7.7.2).

For all modes a search is carried out in scanline order, from left to right, between the parameters of the bounding box within the pre_buffer for identity values i.e. 12, 23, 31. When these are found on any row they are entered into a one-dimensional array that increments the array element i.e. $a[p++]=n$ where n is the value of the pixel x-coordinate. If $p>1$ there must be at least two pixel hits on a scanline producing a span of pixels between them for interpolative shading to calculate their intensities (except

for flat shading modes where the intensities are the same). To find which edges of the perimeter the first and last values are attributed to (*a[0], a[p-1]*), the array is inserted as an element in the pre_buffer row array and the present scanline (*m*) in the pre_buffers column array and the values of those pre_buffer pixel coordinates are interrogated to determine the identities i.e. in pseudocode:

```
if (pre_buffer[a[0]][m]==12) get corner info
if (pre_buffer[a[0]][m]==23) get corner info
if (pre_buffer[a[0]][m]==31) get corner info
.....
.....
if (pre_buffer[a[p-1]][m]==12) get corner info
if (pre_buffer[a[p-1]][m]==23) get corner info
if (pre_buffer[a[p-1]][m]==31) get corner info
```

For flat shading one intensity value fills each triangle so there is no need to interpolate corner intensity values as in the mode of Gouraud shading or vertex normals for Phong shading. Hence, the *corner info,* consists simply of the corner depth values for hidden surface removal.

5.7.7.1 Gouraud shading

Knowing the identity allows the values of the triangle corner pixel y-coordinates, intensity values and depth values to be used for interpolation in the Gouraud modes i.e. for the leftmost line:

```
ia.red=(1/(y1-y2))*((i1.red*(m-y2))+(i2.red*(y1-m)));
ia.green=(1/(y1-y2))*((i1.green*(m-y2))+(i2.green*(y1-m)));
ia.blue=(1/(y1-y2))*((i1.blue*(m-y2))+(i2.blue*(y1-m)));
za_depth=(1/(y1-y2))*((z1*(m-y2))+(z2*(y1-m)));
```

and for the rightmost line:

```
ib.red=(1/(y1-y3))*((i1.red*(m-y3))+(i3.red*(y1-m)));
ib.green=(1/(y1-y3))*((i1.green*(m-y3))+(i3.green*(y1-m)));
ib.blue=(1/(y1-y3))*((i1.blue*(m-y3))+(i3.blue*(y1-m)));
zb_depth=(1/(y1-y3))*((z1*(m-y3))+(z3*(y1-m)));
```

where *il.red,green,blue, i2.red,green,blue* and *i3.red,green,blue* are the calculated intensity values for the relevant corners and *ia.red,green,blue* and *ib.red,green,blue* are the estimated values of the two pixels in question. Similarly, *z1, z2* and *z3* are the depth values of the three corners and *za_depth* and *zb_depth* are the depth values of the pixels. Shading can now be carried out on all pixels in the span by incrementation:

```
dis.red=(ib.red-ia.red)/(a[p-1]-a[0]);
dis.green=(ib.green-ia.green)/(a[p-1]-a[0]);
dis.blue=(ib.blue-ia.blue)/(a[p-1]-a[0]);
```

the values *dis.red,green,blue* are the incrementing values for each adjacent pixel in the span and the values are arrived at simply by dividing the difference in intensities of the two pixels by the number of pixels between them. Similarly, the depth values for each pixel are incremented:

```
dz_depth=(zb_depth-za_depth)/(a[p-1]-a[0])
```

Each pixel depth value is compared with the pixel value in the z-buffer and if it is less than that value, *with respect to the projection point in use at the time* (section 5.8), then it overwrites that value and becomes the new depth value until it may also be overwritten in later incrementations. If the present depth value does overwrite the earlier value then the incremented intensity can now be set as the intensity value of that pixel in the frame buffer.

5.7.7.2 Phong shading

In exactly the same way that the estimated intensity values were calculated for the two pixels on a row in the previous sub-section, the vertex normals for those pixels are estimated from the pre-calculated vertex normals at the triangle corners:

```
Na.x=((vN1.x*(m-y2))+(vN2.x*(y1-m)))/(y1-y2);
Na.y=((vN1.y*(m-y2))+(vN2.y*(y1-m)))/(y1-y2);
Na.z=((vN1.z*(m-y2))+(vN2.z*(y1-m)))/(y1-y2);
Nb.x=((vN1.x*(m-y3))+(vN3.x*(y1-m)))/(y1-y3);
Nb.y=((vN1.y*(m-y3))+(vN3.y*(y1-m)))/(y1-y3);
Nb.z=((vN1.z*(m-y3))+(vN3.z*(y1-m)))/(y1-y3);
```

Again, similar to the intensity estimations of Gouraud shading, an incremental normal value is calculated for the pixels in the span and added or subtracted from the linear estimation of the leftmost line's pixel in question. If the interpolated depth passes the test of the z-buffer then a function that calculates and adds together all intensity components of the Phong illumination model is called to provide the intensity value for that pixel. This obviously is much slower than Gouraud shading and it is generally known in standard computer graphics to take up to as much as 50% of the processing time.

Within this thesis when using experimental programs the output file format has been the ppm format (portable pixmap). This may now be displayed via a suitable graphics viewer onto an LCD fitted with a lens array with the same geometries as the virtual lens array and the lenslets will allow the pixels to act as if they have directional properties. Thereby an integral image is formed.

5.8 Program flow considerations

In a real integral imaging camera the intensity values at the recording media are obviously additive and to replicate this, in this model, it is necessary to allow the hidden surface removal technique to act separately for each projection of the whole scene from each projection point. Each subsequent scene projection must only compare depths for hidden surface removal with the present scene projection and then

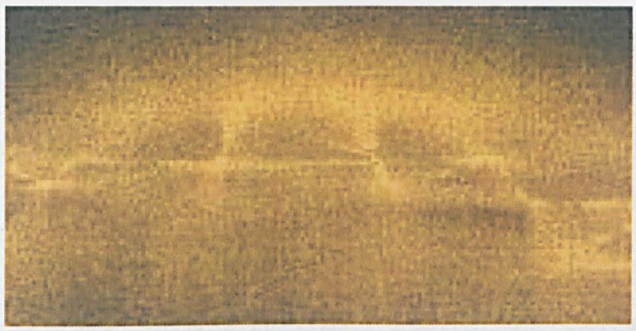

Figure 5.9: Piecewise linear intensity changes across triangular distribution boundaries

add to the pixel intensity already present at each pixel from earlier scene projections. Dividing the resulting pixel intensities by the number of projection points provides the average intensity and keeps the values within the allowed top limit (i.e. 255). To accommodate this the z-buffer must be re-initialised after each scene projection. It is also necessary to provide two frame buffer arrays, a sub-image buffer to capture pixel intensities that filter through the z-buffer for each scene projection, which is subsequently re-initialised after each one is completed, and a compositing buffer to add and store the distribution result of each successive projection. Extra storage space and increased processing time continually to re-initialise the z-buffer and the sub-image buffer is therefore required to accurately use additive intensity.

The alternative is to configure the program flow such that each triangle is taken separately and projected from the sequence of projection points in turn, but many problems exist when additive intensity values are used. As an encoding triangle additively overlaps itself during the projection sequence strong Mach banding effects result that generate piecewise linear intensity changes across the triangle's distribution boundary (Figure 5.9). The reason for this is that as each encoded triangle is shifted from its predecessor due to the shifted projection point location the intensity builds up centrally, but around the encoded edge only one hit may be recorded. In this scenario, depending upon scene complexity, a kaleidoscope of colour, bearing no resemblance to the initial scene colour, is often the outcome (Figure 5.10). Obscured objects have their intensities incorrectly added to the final image, and in most cases, whatever intensity averaging technique is used, produce transparency effects. These are obviously more noticeable if the 'obscured' object has a more dominant colour than the object producing the obscuration. Hence, even by dividing the final intensities of

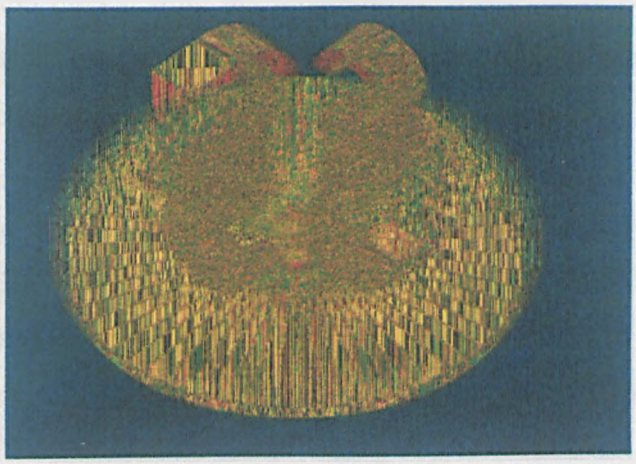

Figure 5.10: Incorrect program flow generates unusual LeSDs when using additive intensity

the pixels by a pixel hit counter the transparency problem is not solved. However, the problem of a kaleidoscopic colour distribution is dealt with.

The intensity distribution still produces an integral image when decoded by a lens array; this signifies that the spatial distribution is the criteria for creating a volumetric image, not the intensity values. A further confirmation of this is that a totally homogeneously matt black object when integrally captured and displayed will replicate the original volume of the object. The effort expended in reaching perfect intensities should not be one that is concerned with creating a volumetric image but one that has, as its aim, accurate colour replication. *Where* the intensity resides at the recording media, not *what* the intensity is, is the most important criteria with respect to actually generating the 3D aspect of an integral image. A less accurately defined volumetric representation of a scene in terms of colour replication can therefore be accomplished inexpensively by using the alternative program flow but without additive intensity. This approach requires less storage for saving triangle perimeter coordinates and ambient and diffuse components as these can be calculated and used on a triangle-by-triangle basis.

In the computer generation of integral images increasing the number of projection points and using additive intensity produces anti-aliased LeSDs (see sub-section 6.3.3). This method can be used in the more expensive program-flow technique when processing time and storage is no problem. However, by capturing an LeSD at a

higher resolution and then scaling the image down equivalently by combining pixels and averaging intensities also provides anti-aliased images and this technique is suitable for the alternative program flow.

The following two blocks of pseudocode show the differences between additive intensity (1st block) and single intensity (2nd block) models. They both show the programs using Gouraud shading with the applications of the other shading techniques in comments. Animation loops and scene transformations are ignored but the initial scale is included. The stages of the transforming object vector data are depicted as V1, V2, V3 etc.

```
procedure Gouraud_IntegralRendering()
  Input_object();
  Scale(scale_factor,V1[i]);
  Compute_norms_cull(V2[i]);
  Compute_vertex_norms(norm[i/3],V3[i]);
  Draw_perimeters_Store(V4[i]);
  Compute_Ambient_Diffuse_Store(vertex_norm[i]);
  (Flat: (norm[i/3]) Phong: N/A)
  foreach (projection_points from first to last)
    foreach (triangle_vertices from first to last)
      if Compute_image_coords(V4_perimeter[p],projection_vector,V4[i])=True
        Compute_specular(vertex_norm[i],projection_vector,light_vector);
        (Flat: (norm[i/3]) Phong: Computes all components)
        Compute_fill(depth[3],intensity[3],y_pixel_coord[3]);
        (Flat: (depth[3],intensity)
         Phong:(depth[3],vertex_norm[i],y_pixel_coord[3]))
      end
    end
  end
 Display_image();
end
```

Pseudocode: Block 1

```
procedure Gouraud_IntegralRendering()
  Input_object();
  Scale(scale_factor,V1[i]);
  Compute_norms_cull(V2[i]);
  Compute_vertex_norms(norm[i/3],V3[i]);
  foreach (triangle_vertices from first to last)
    Draw_perimeter(V3[i]);
    Compute_Ambient_Diffuse(vertex_norm[i]);
    (Flat: (norm[i/3]) Phong: N/A)
    foreach (projection_points from first to last)
      if Compute_image_coords(V3_perimeter[p],projection_vector,V3[i])=True
        Compute_specular(vertex_norm[i],projection_vector,light_vector);
        (Flat: (norm[i/3]) Phong: Computes all components)
        Compute_fill(depth[3],intensity[3],y_pixel_coord[3]);
        (Flat: (depth[3],intensity)
         Phong:(depth[3],vertex_norm[i],y_pixel_coord[3]))
      end
    end
  end
 Display_image();
end
```

Pseudocode: Block 2

5.9 Conclusions

It is worth the effort taken to develop these programs if only to show that integral images can be generated and *rendered* by a forward geometric projection technique that is a software model for a real finite-sized aperture integral imaging camera. The control over program flow is possible as the programs are written from 1st principles allowing the freedom to experiment. It is difficult and expensive to obtain lenslet and LCD pixel pitches that give an integer number of pixels per lens and the lens array has to be manufactured specifically to dimensions of a particular pixel pitch and hence to a particular LCD. The model based on a finite-sized aperture allows a non-integer number of pixels per lenslet to be employed and as such enables lens arrays to work on any LCDs to good effect. To the author's knowledge this method is an original contribution in the area of the computer generation of integral images and it is innovative as an idea, and in the design and implementation.

Scene transformations and animations are a bonus to this research and were evolved, like all of these programs, from the inspiration received when the first spot image in

black and white, generated from the minimalist, non-rendering program of Chapter 3, displayed an image that had *real* depth, without the use special glasses.

Another methodology yet to be explored is the 3D-from-2D model that is the second method devised in Chapter 2. In comparison to the finite-sized aperture model the 3D-from-2D, or pinhole model, is a much simpler task to develop but a real difficulty is seen to be the goal of real-time. The technique is simple to implement because by taking out the lens array and implementing other modifications use can be made of the code explained in this chapter. Realisation of real-time processing is difficult because specialised hardware needs to be developed and this is outside the remit of this thesis for reasons of time and money.

Forward Projection Pinhole Rendering Model

6.1 Introduction

The forward projection pinhole rendering model is termed the 3D-from-2D model in sub-section 2.3.1 because standard 2D forward projections of a scene from different viewpoints are captured as sub-images that are composited in such a way as to produce a LeSD (figure 6.1). This technique, as explained in Chapter 2, is equivalent to modelling a lens array wherein each lenslet acts as a pinhole camera and rays are thus only allowed to pass through the vertex (pinhole position) of each lenslet. Modifying the program already written in the previous chapter is an obvious step to allow full control over every aspect of rendering. This is seen as a logical approach because it will prove that an integral image can be generated in this way, and if successful in practice then many speedups can be used for a real-time integral renderer.

6.2 3D-from-2D technique

Sub-section 2.3.1 shows that each sub-image's *N*th pixel column, starting at the first sub-images 1st pixel column, the 2nd sub-images 2nd pixel column etc. etc. are the

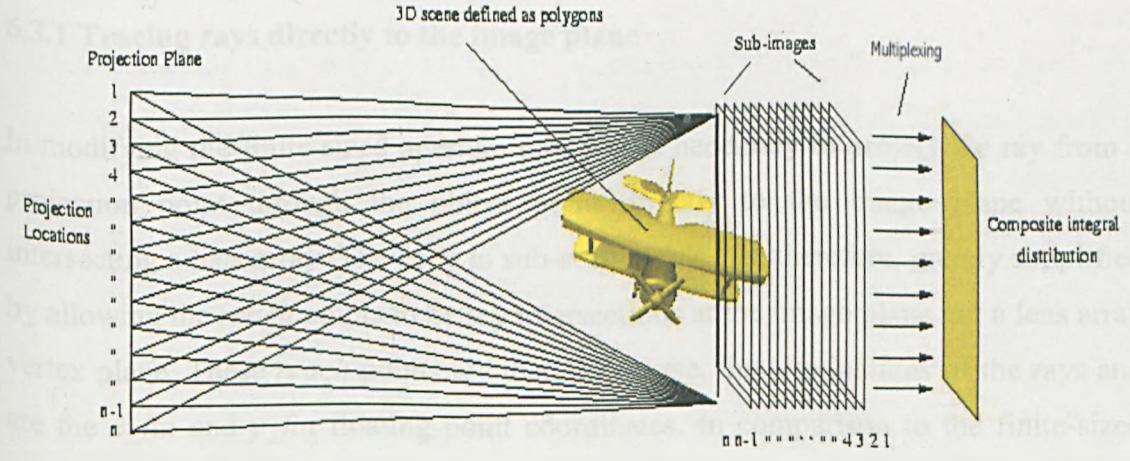

Figure 6.1: LeSD generation by multiplexing 2D projections for pinhole model

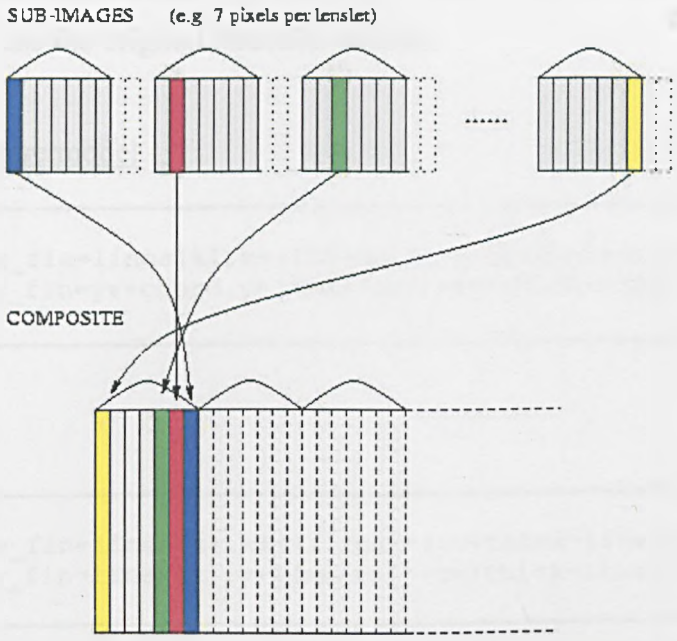

Figure 6.2: Multiplexed composite formed from multiple sub-images

required pixel values to be placed into the composite image for a pinhole lens array model. Figure 6.2 shows that these columns are actually inverted behind each lenslet and this is due to the direction that the projection points follow each other. If the columns were put into the composite image without inversion (or the sub-images were generated from projection points processed from the other direction) then a pseudoscopic integral image would ensue.

6.3 Compositing sub-images in software

6.3.1 Tracing rays directly to the image plane

In modifying the finite-sized aperture model it is necessary to project the ray from a projection point through the object point directly to the image plane without intersecting a lens array. The code in sub-section 5.7.4 is, therefore, greatly simplified by allowing the *reach* points to be ray intersections at the image plane not a lens array vertex plane. These *reach* points are then, of course, the terminations of the rays and are the *x_fin* and *y_fin* floating point coordinates. In comparison to the finite-sized aperture model of sub-section 5.7.4, the pinhole lenticular perspective mode's *y_fin*

does not contain an intermediate y-position (*yv+chord.y*) at the lenslets curved surface and continues to use the original direction cosines:

Finite-sized aperture model

```
x_fin=lines[k].x+((ll/nn)*(zv+thick-lines[k].z));
y_fin=yv+chord.y+((mm2/nn2*(zv+thick-chord.z)));
```

Pinhole model

```
x_fin=lines[k].x+((ll/nn)*(zv+thick-lines[k].z));
y_fin=lines[k].y+((mm/nn)*(zv+thick-lines[k].z));
```

The projection locations on the aperture are set at the mid-point of the parallel groups. The locations of these can be determined by extrapolating lines from the centre of each pixel of the central lenslet in the lens array, through the pinhole, and on to the aperture. Where they meet at the aperture determine the location of the projection points for the case of an odd number of lenslets, for an even number the same technique can be performed on an imaginary lenslet centrally straddling the two centre lenslets of the virtual array. As explained in sub-section 2.3.1 the translation of the scene to the image plane from a given projection location is equivalent to performing a perspective transformation to obtain a distorted scene in 3D space and then to orthographically project the result onto the same plane.

The requirement is to keep check of the current projection point in use (*ka*) and write code within the shading procedure to select which pixels and pixel intensities are to be written to the final composite image. It is at this level of control that many commercial or open source graphics programs fail to allow straight-forward intensity and pixel manipulations and those that do only enable stereo imaging.

6.3.2 Software multiplexing filter

Several approaches to producing a software filter were considered. Two were examined and tested. It was realised that to produce and hold each sub-image in memory while waiting for all sub-images to be generated is inappropriate, especially when considering that each sub-image generated from each projection point is the same size as the final composite image. Consequently, it is convenient to generate one sub-image at a time and then transfer the required pixel intensities to the composite buffer, and reinitialise and reuse the sub-image buffer for the results of further scene projections.

Another possible method is to place a filter within the shading interpolation algorithm only allowing the correct pixel intensity values to go direct to the composite buffer pixels, depending upon the projection point in current use. This is on-the-fly multiplexing, it saves memory and is efficient in the sense that as the pixel coordinate and *view* is the criteria for admission (not intensity) then it is not necessary to calculate intensities across a span if that span is not to be included in the final image. The compositing technique is, after all, selecting whole columns for admission and columns are, in this program, conveniently in the raster direction (sub-section 3.4.2) i.e. the x-direction. Furthermore, the sub-image buffer is not required as the selected pixel intensities are sent directly to the composite buffer.

In designing the filter a visual reference can be constructed to show the pixel columns in each sub-image that are required to be routed into the relevant columns in the composite. Figure 6.3 and Figure 6.4 shows a section of a lens array (sub-images) with pixel columns going into the page and numbered. In these examples there are 7 pixels per lenslet and 5 pixels per lenslet respectively.

The first column of pixels, in the 7 pixels per lenslet example, begins at number 21 (the first column off page begins at zero) and all the first columns of each lenslet belong to the first sub-image, similarly, all the second columns behind each lenslet belong to the second sub-image .... etc. etc. It is required that pixel intensities of column 21 are routed to column 27 in the composite along with 28, 35 and 42 to 34, 41 and 48 respectively. These are all the first sub-image columns. Similarly for the

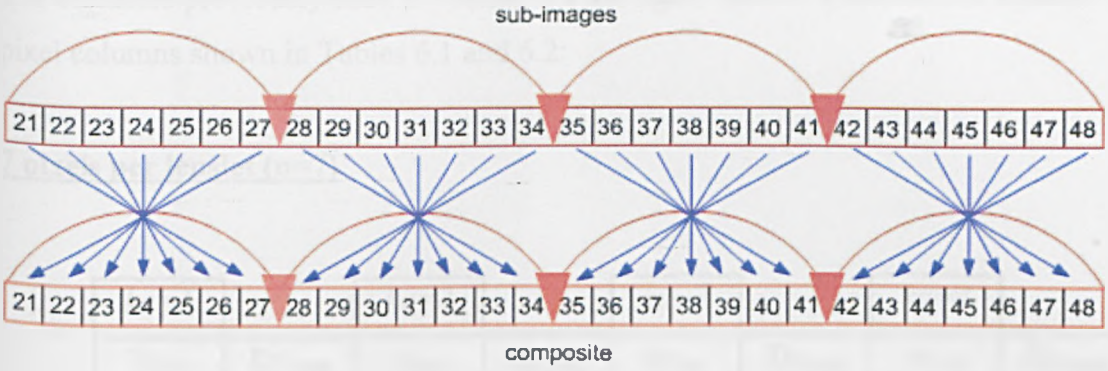

Figure 6.3: multiplexing with 7 pixels per lenslet

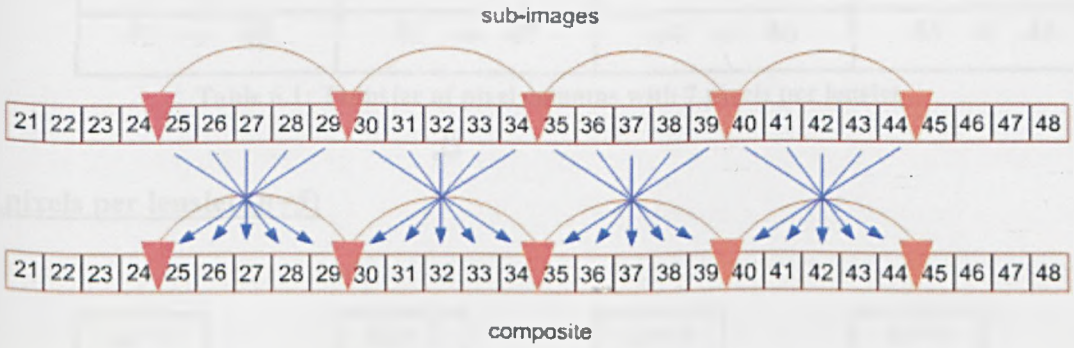

Figure 6.4: multiplexing with 5 pixels per lenslet

---

second sub-image 22, 29, 36, and 43 are transferred to 26, 33, 40 and 47 columns in the composite .... etc. etc.

The first 4 columns related to the first 4 sub-images of both systems are shown in Figs 6.3 and 6.4 and Tables 6.1 and 6.2, revealing the composite column values.

The complexity in deriving an equation (and hence code) to give these transferred results and enable it to work for all systems initially seems to present a difficult task, but surprisingly, the resulting derived equation looks very simplistic:

$$Kp_{comp} = kp_{sub} + n + 1 - 2ka$$

The variables previously used in section 2.4 are again used to describe this transfer of pixel columns shown in Tables 6.1 and 6.2:

7 pixels per lenslet (n=7)

ka=1		*ka=2*		*ka=3*		*ka=4*	
kp_{sub}	Kp_{comp}	kp_{sub}	Kp_{comp}	kp_{sub}	Kp_{comp}	kp_{sub}	Kp_{comp}
21 →	**27**	22 →	**26**	23 →	**25**	24 →	**24**
28 →	**34**	29 →	**33**	30 →	**32**	31 →	**31**
35 →	**41**	36 →	**40**	37 →	**39**	38 →	**38**
42 →	**48**	43 →	**47**	44 →	**46**	45 →	**45**

Table 6.1: Transfer of pixel columns with 7 pixels per lenslet

5 pixels per lenslet (*n=5*)

ka=1		*ka=2*		*ka=3*		*ka=4*	
kp_{sub}	Kp_{comp}	kp_{sub}	Kp_{comp}	kp_{sub}	Kp_{comp}	kp_{sub}	Kp_{comp}
25 →	**29**	26 →	**28**	27 →	**27**	28 →	**26**
30 →	**34**	31 →	**33**	32 →	**32**	33 →	**31**
35 →	**39**	36 →	**38**	37 →	**37**	38 →	**36**
40 →	**44**	41 →	**43**	42 →	**42**	43 →	**41**

Table 6.2: Transfer of pixel columns with 5 pixels per lenslet

The filter that checks each projection point number and each column, for transfer and admission to the composite, must also be derived. Again, a complex deduction but a simple expression is the result:

If the remainder equals zero when kp_{sub}-ka+1 is divided by n then those sub-image pixel intensities are calculated, interpolated, and finally transferred to their new pixel coordinates in the composite.

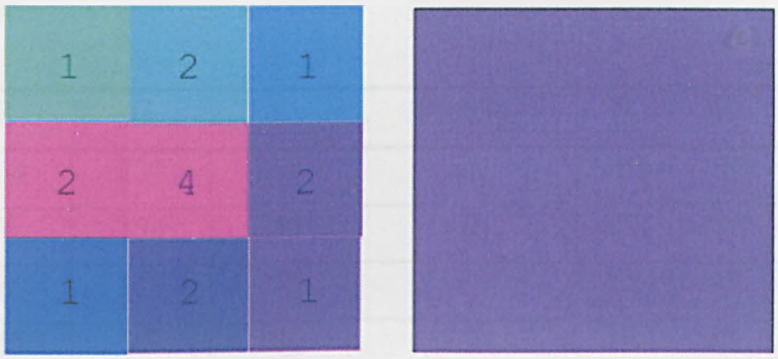

Figure 6.5: Weighted super-pixels and resultant pixel intensity

6.3.3 Anti-aliasing

Just as anti-aliasing techniques are available for standard computer graphics they can be used for integral imaging but at a cost that may be too expensive with respect to processing time for real-time generation. During capture a single point of a scene is scattered amongst many lenslets and then is reintegrated during playback, ideally to a single discrete position in image 3D-space, so it is arguable that integral images are inherently already anti-aliased.

Super-sampling is one method whereby each scene projection is captured on a pixelated plane that has a higher resolution than the final display resolution. This method increases the number of 'virtual' pixels behind each lenslet and hence the number of equivalent projection points. The pixels are then averaged by digital convolution to the final lower display resolution. If, for example, the images are captured at 3 times the display resolution then each display pixel will have an intensity value made up of nine super-pixel values. These nine pixel values can first be weighted before adding and averaging [58].

Figure 6.5 and the Table 6.3 shows an example of super-pixel intensities, weighting values and resultant pixel intensity after summing the products and averaging:

In raster order:

R	G	B	Weight	R	G	B
104	237	255	×1	104	237	255
76	194	255	×2	152	388	510
69	143	255	×1	69	143	255
255	25	180	×2	510	50	360
255	31	228	×4	1020	124	912
171	87	255	×2	342	174	510
130	129	255	×1	130	129	255
71	37	255	×2	142	74	510
171	87	255	×1	171	87	255
			Sum =	2640	1406	3822
Average by 16 for resultant pixel intensity =				**165**	**88**	**239**

Table 6.3: Weighting, adding and averaging of super-pixels for super-sampling

The super-pixels are created in the rendering chain after the translation of each scene triangle to the image plane. Setting the OUTPUT_RES value in the object file controls the resolution and this sets the variable *dpi* that is used during the pixelation process i.e. `sc=dpi/25.4` (sub-section 5.7.4). Each scene in turn must be stored before any transfer to the composite and in traversing the blocks or windows of nine super-pixels the buffer containing the pixel intensity values must jump 3 super-pixels along the row and column to settle on the next central super-pixel to compute the next sum of products. From the weighted example in Figure 6.4 and the table it is clear that the final intensity value of each window after summing the products is divisible by 16 not 9. The computational overhead obviously increases for windows containing say a 25 (5x5) or a 49 (7x7) filter kernal and the wider the kernal the better it is at reducing high frequency aliasing artefacts but at the cost of increased blurring.

A non-weighted approach is to increase the number of projection locations equally in each row on the aperture (or the single row when using semi-cylindrical lens arrays), this increases the number of hits per pixel from 1 to a number directly proportional to

the number of projection points. That is to say, increasing the frequency of the projection points by a factor of *n* gives each pixel *n* hits from slightly different parts of the scene. These hits are then added and averaged. It is obvious that to keep this method simple *n* must be an integer number.

Integrating this novel anti-aliasing method into the multiplexing code requires that the current projection point number (*ka)* is correctly assigned its composite pixel coordinates and that the filter uses the correct values when considering that each pixel receives multiple hits from different projection locations. This is achieved by a sequence of *if* or *case* statements relating to the projection point number whereby each group of *n* projection points per pixel are given their group identifiers (`ka_group`):

```
if ((ka>0)&&(ka<=(proj_points/n)))
  {
   Kp_comp=kp_sub+n+1-(2*ka);
   ka_group = 1;
  }

if ((ka>(proj_points/n))&&(ka<=((proj_points*2)/n)))
  {
   Kp_comp=kp_sub+n+1-(2*ka);
   ka_group = 2;
  }

if ((ka>((proj_points*2)/n))&&(ka<=((proj_points*3)/n)))
  {
   Kp_comp=kp_sub+n+1-(2*ka);
   ka_group = 3;
  }
if ..........
   ..........
   .......... etc. etc.
```

and this identifier is used in the filter instead of the projection point number:

```
if (((kp_sub-ka_group+1)%n==0))
```

The variable `n` in the code, as usual, is the number of pixels per lenslet and should not be confused here with the factor *n* that the projection points are increased by. The

total number of projection points used (an integer multiple of the number of pixels per lenslet) is contained in the variable `proj_points` and is set by the user in the object file under PROJ_LOCATIONS. The pixel intensity values for each pixel are added as the values pass the filter and z-buffer tests and are then finally averaged.

6.4 Conclusions

Integral images generated using this pinhole procedure when compared to those generated by the finite-sized aperture model are almost indistinguishable. Integral objects in the latter tend, if anything, to be slightly larger giving the impression that the pinhole integral objects look slimmer. The probable cause of this is spherical aberration that is inherent in the finite-sized aperture model causing image point spread at the capture plane.

The different types of integral forward projection techniques devised and implemented so far can now be put to use in various creative endeavours ranging from integral animations on high and low resolution displays to integral projection. These activities should bring further insights and knowledge to the computer generation of integral displays by using the full potential of these programs.

Integral Image Production and Display Experiments

7.1 Introduction

To quantify the abilities of computer-generated integral images and push their inherent characteristics to the extremes a variety of experiments were carried out, observations made, and concepts clarified.

7.2 Integral image animations

A sculptor [59] developed and choreographed a torii sequence to realize a cyber-structure modelled in 3DS Max and the first frame was saved as a VRML2 file. When analysing the contents of the file it is found that there is one object (torus01), which is compressed using IndexedFaced sets. A remaining three objects are manufactured from this one object by scaling and mirroring. When the original object is mirrored (torus02) both torus01 and torus02 are instantiated and scaled down to half the original size (torus03, torus04); thus forming four objects.

The animation details are:

Torus01 and torus02, as a pair, are rotated in opposite directions and over 400 frames are scaled down to half-size. Similarly, the remaining pair are rotated but scaled up to the original size. For the next 400 frames the reverse process is performed completing the animation; the last frame being the same as the first. The torii are allowed to interpenetrate each other as they rotate, scale up and scale down, producing continuously changing shapes and forms allowing the viewer visual accessibility of the interior of the scene.

The VRML2 file was transformed to the uncompressed integral imaging format. A program was written (see Appendix G, Part 1) to perform the parsing, uncompressing and file conversion to produce a single object (torus01). A further program was

written to take the object vertices and scale and mirror them where necessary producing the initial state of the scene. In addition, a further program was written to recursively calculate the scaling factor required for the torii during the production of each frame. To scale down one pair of torii to half their size over 400 frames the factor is found to be –0.001253 and to scale up the half size pair to full size the factor is +0.002306. Within the animation the separate torii are required to revolve around the common global centre not around there own local centres. The torii are pre-morphed during their creation in 3DS Max and are not symmetrical in shape, on rotation around their own centres they move apart from one another by a considerable distance. Revolving the torii around the global centre produces the required effect of inter-mingling.

Given that over the cycle of the animation the scene rotates 360° the rotation factor for 400 frames is simply 180/400. A negative and positive value is given for the torii of each pair. This rotation is set about the y-axis. Other parameters to initiate are the aperture size, depth of virtual lens array within the scene, virtual lens array parameters, material finish and the output resolution.

Instead of generating a batch of 800 frames directly from the integral renderer 400 sequences were first generated and these same frames used to provide frames 401-799. This was accommodated using a script file. The method used was to copy frame 399 to frame 401 then frame 398 to frame 402 etc. until frame 1 is copied to frame 799. These 799 frames were then encoded to an MPEG2 file and run on the Samsung LCD. This LCD has a resolution of 96.212dpi and a pixel pitch of 0.264mm. The lens array used has 12 lines per inch giving an integer number of 8 pixels behind each lenslet. Although the integer number of pixels per lenslet prevents moiré interference being generated between the regular structure of the lenticular screen and that of the pixel grid, colour moiré generated by the RGB sub-pixels was still present. Using a dark colour for the scene and a black background minimized this effect.

The integral renderer was set-up to allow additive intensity and as such produced 3 intensity values for each pixel. These values were then averaged to provide an anti-aliased rendered animation. The scene file is presented in Appendix G, Part 2.

The animation ran correctly but the holographic-like effects are difficult to reproduce in front of the screen at this low-resolution although depth effects behind the screen plane are more clearly discerned [60]. The same animation was generated for the high-resolution T221 IBM LCD (QUXGA – 3840x2400) that has 9.2 million pixels with a pixel size of 0.124mm. The best that can be achieved on this LCD, when considering that each frame file takes up 27Mb of hard drive space, is to display the separate consecutive frames as fast as possible (see 8.2.1) which is approximately one frame per second. Nevertheless, the resultant display demonstrates due to the high resolution and small pixel size, for the first time on any LCD, the full benefits of computer generated integral imaging. When the viewer places a hand close to the screen it is possible to see parts of the scene not occluded by the hand reformed in front of the hand. In the first frame the virtual lens array is set at 50% of the scene depth and the volumetric image extends 12cm in front and 12cm behind the screen yet still retaining sharpness, focus and intensity due to the information density. Some frames, during the sequence, move further out from the screen plane and image cohesion is lost around 15cm although for all frames the scene behind the screen plane remains volumetrically stable and complete. A possible reason that contributes towards this difference in resolving power between points in front and points behind the array is that the lenslet coverage for points in front of an array is less than the coverage for points behind the array. This is established in the following section.

7.3 Comparison of coverage between front and rear array points

Figures 7.1 and 7.2 are graphs that are produced using an equation that calculates the number of lenslets covered for different depths:

$$\text{Number of lenslets covered} = \frac{Az_{pt}}{P(d - z_{pt})}$$

In this example $d = 562.6mm$, $P = 0.5992mm$ and $A=194mm$.

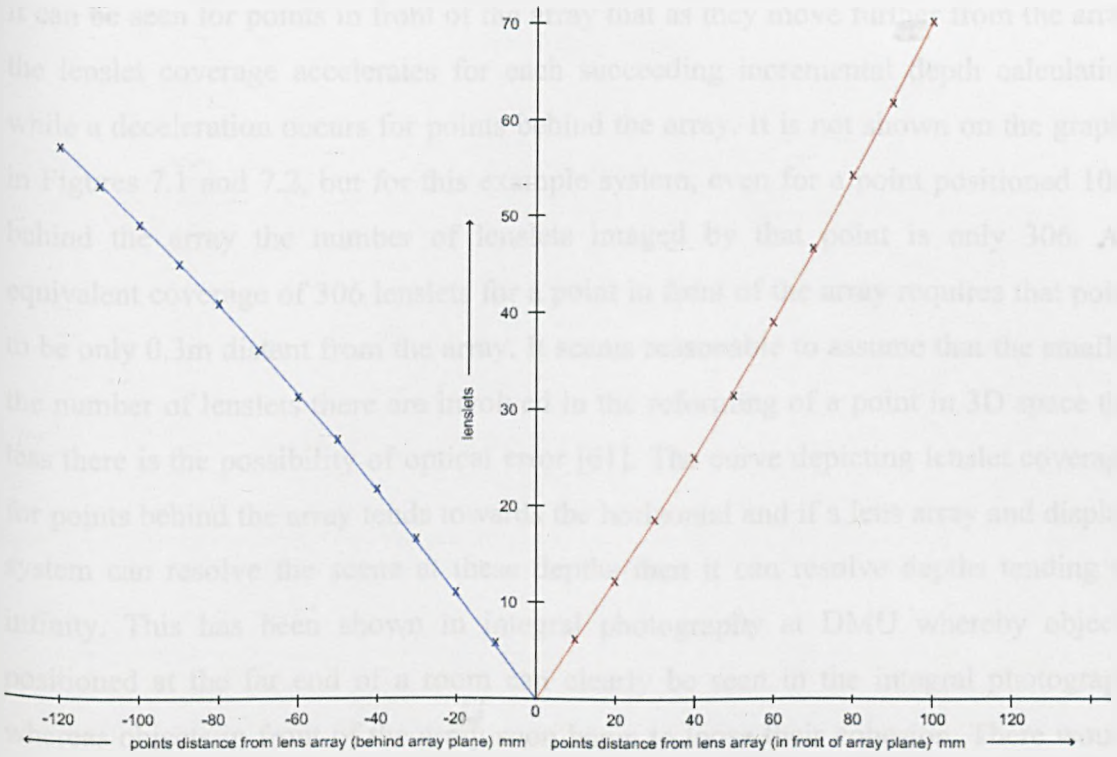

Figure 7.1: Graph showing comparison of lenslet's coverage for points in front and points behind the array

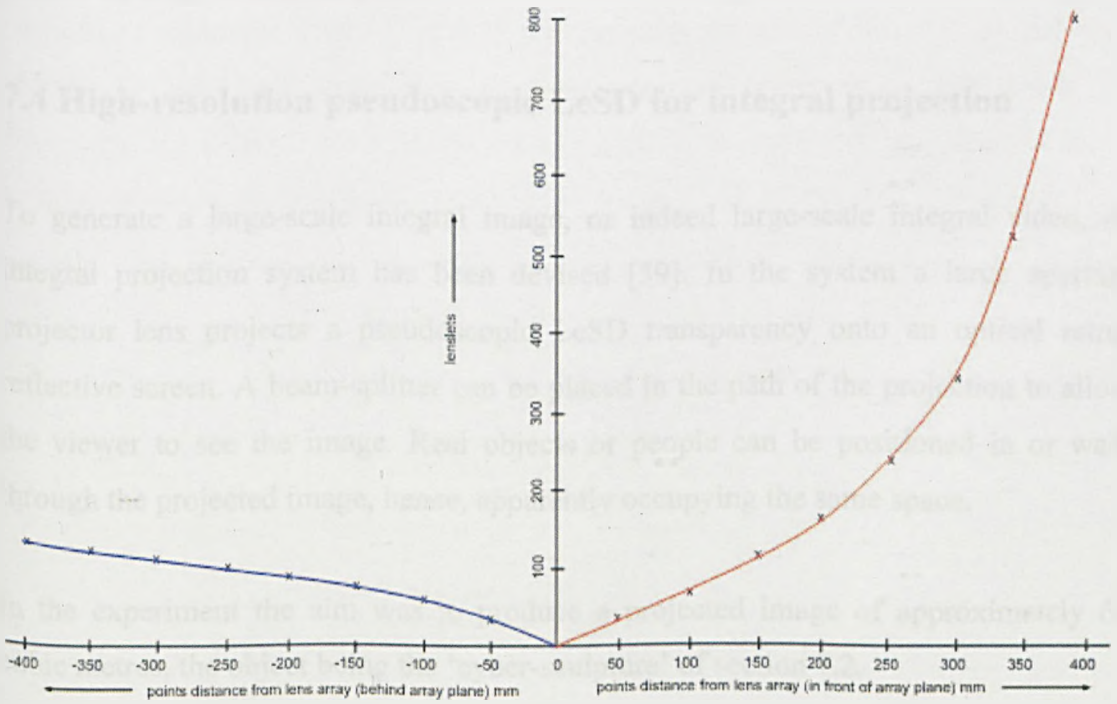

Figure 7.2: Extended graph showing curve tendencies towards horizontal and vertical directions

It can be seen for points in front of the array that as they move further from the array the lenslet coverage accelerates for each succeeding incremental depth calculation while a deceleration occurs for points behind the array. It is not shown on the graphs in Figures 7.1 and 7.2, but for this example system, even for a point positioned 10m behind the array the number of lenslets imaged by that point is only 306. An equivalent coverage of 306 lenslets for a point in front of the array requires that point to be only 0.3m distant from the array. It seems reasonable to assume that the smaller the number of lenslets there are involved in the reforming of a point in 3D space the less there is the possibility of optical error [61]. The curve depicting lenslet coverage for points behind the array tends towards the horizontal and if a lens array and display system can resolve the scene at these depths then it can resolve depths tending to infinity. This has been shown in integral photography at DMU whereby objects positioned at the far end of a room can clearly be seen in the integral photograph whereas objects in front of the array soon begin to loose their cohesion. There would of course be no necessity for a scene to be resolved at infinity for points in front of the display as the viewers are looking at scenes in front of them, but as the curve is tending to the vertical this would not be a possible consideration anyway as it would require an array of infinite size.

7.4 High-resolution pseudoscopic LeSD for integral projection

To generate a large-scale integral image, or indeed large-scale integral video, an integral projection system has been devised [59]. In the system a large aperture projector lens projects a pseudoscopic LeSD transparency onto an optical retro-reflective screen. A beam-splitter can be placed in the path of the projection to allow the viewer to see the image. Real objects or people can be positioned in or walk through the projected image, hence, apparently occupying the same space.

In the experiment the aim was to produce a projected image of approximately 64 cubic metres, the object being the 'cyber-sculpture' of section 7.2.

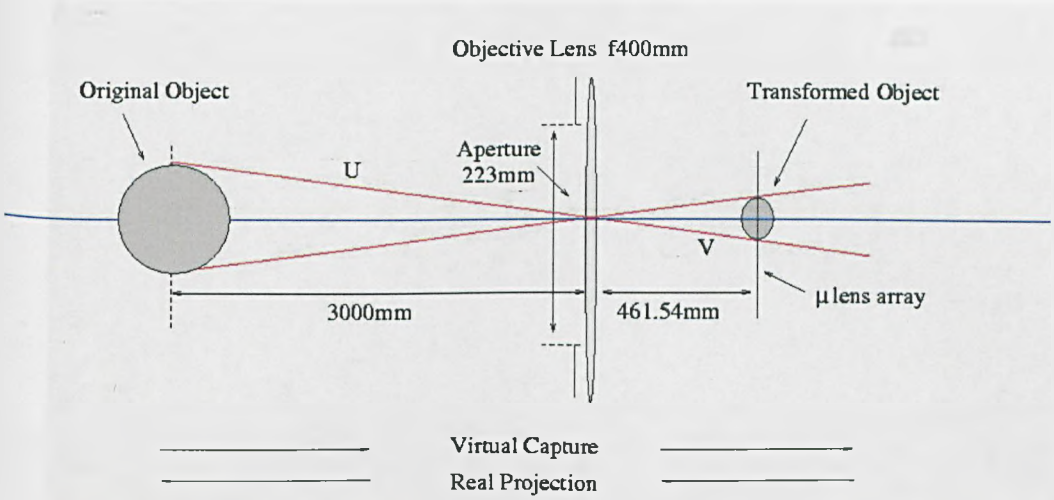

Figure 7.3: Set-up parameters of the projection system showing the pre-processing transformation required to be performed on the original object data

To ensure that the projected image is correctly scaled to life-size the scene data has to be transformed by a virtual objective lens with the same parameters as the real projection lens (figure 7.3). The effect is to produce a scaled down object that has a lower width to height ratio than the original i.e. spatially compressed along the optical axis. The virtual objective lens is generated using cosine directions and the lensmaker's equation where U and V are the skew lengths of 'rays' from polygon vertices to lens and from lens to final position respectively. The value of *objectivelens_z* is the scene's global centre added to the objective lens distance from the original object i.e. 3000mm:

```
incident_vec.x=-x1;
incident_vec.y=-y1;
incident_vec.z=objectivelens_z-z1;
U=vect_mag(&incident_vec);
V=((f*length)/(length-f));
ll=incident_vec.x/length;
mm=incident_vec.y/length;
nn=incident_vec.z/length;
x1=ll*planar_diag2;
y1=mm*planar_diag2;
z1=nn*planar_diag2;
```

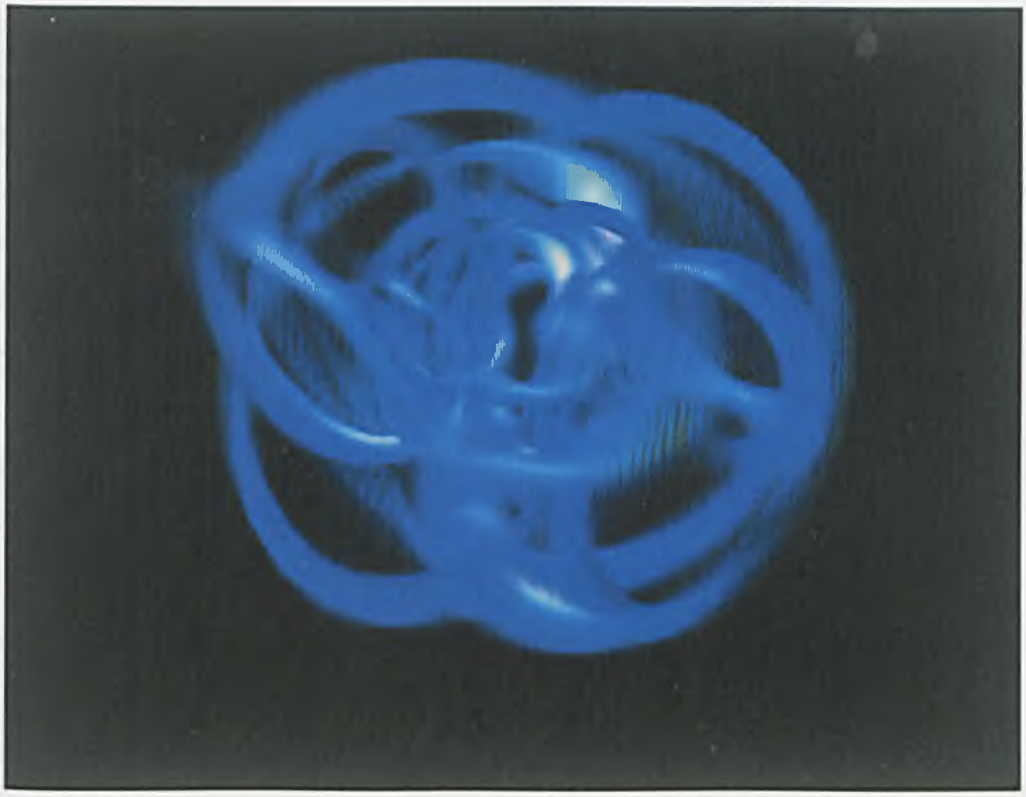

Figure 7.4: High-resolution pseudoscopic LeSD printed as a transparency for projection

When producing an image of the transformed object, rotated by 90°, it is noticed that the scene is scaled down and spatially compressed along the optical axis showing that the virtual objective lens is correctly represented in code. This compression is automatically measured within the program and was found to be correct when compared to hand calculations. The images must be pseudoscopic in order that after projection they are viewed orthoscopically. This means that on replay through a lens array they are pseudoscopic. This is the opposite of what is required for direct viewing and the necessary code has to be modified by interchanging pixel intensity values behind each lenslet to produce this effect. The LeSD was produced for a high-resolution printer (1270dpi) and printed on a Fire 800 printer as a transparency (figure 7.4).

Low-resolution images (300dpi) were produced from the high-resolution images by scaling the LeSDs down by a factor of $300/1270 = 0.236220472$ and these were printed on transparencies on a Minolta colour printer. The low-resolution images were

checked for accurate replay and were found to be sharp integral images with strong holographic-like effects in front of the display. They replay pseudoscopically as expected. The transition to side viewing lobes also occurred simultaneously. However, they were not capable of resolving the depth to an appreciable level on the integral projection system due to their inherent lack of information density leading to ill formed image intensities. The projection of the 1270dpi LeSD, however, is quite impressive creating a visible resolved depth of approximately 4m.

7.5 Greater depth resolution by scene compression

An alternative technique of compressing the scene along the optical axis before rendering, to give the effect of greater depth resolution, can be accomplished by artificially positioning the virtual image plane at a greater distance than the focal plane from the virtual lens array.

Figure 7.5 shows the capture aperture is set at a size and distance that is correct for an image field occurring at the back of the central lenslet but the virtual lenslets are given an f-number twice that of the real lens array i.e. an artificial focal plane is set up, in this example, at twice the distance of the real focal plane. Therefore the lenslet coverage for each point in the scene is half that required for the real positioning of the aperture.

During replay, the real lenslet pinholes are now half the distance from the pixels and the pixels angles to the pinholes are thereby much greater; this is reflected in the angles α and β in figure 7.5. The resultant effect is that the volume of the regenerated scene is compressed along the optical axis. Comparing an original object point to the replayed object point position in figure 7.5 can see this. The amount of parallax present is less than it would have been had the aperture been set in the real aperture position but due to the compression and the less lenslet coverage during capture for each point, the depth resolution of the replayed volumetric image is greatly increased. This increase is therefore both apparent and real.

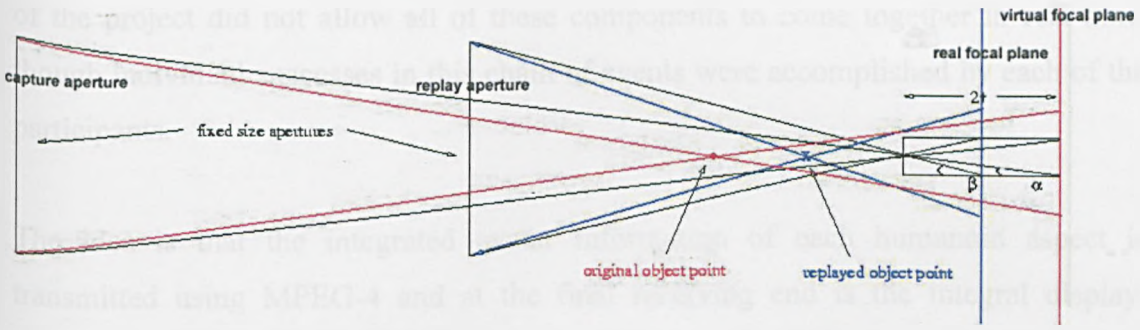

Figure 7.5: Effect of compressing the scene volume along the optical axis by capturing at a longer virtual focal plane than that of the real focal plane

The key feature in this novel technique is that both capture and replay apertures are the same dimensions yet different distances from the lens array. One fills the image fields for a displaced virtual focal plane and the other replays the image fields for the real focal plane.

7.6 Texture-mapped avatars

The representation and simulation of virtual humans (avatars) [62][63] has a range of applications such as TV productions, fashion, computer games, telecommunications, advertising and research in such areas as the car industry, medicine, conferencing and navigable virtual world chat rooms on the Web, to name a few. The research area of avatars is very large and growing and draws on many different research topics. The Prometheus project [64] is an area that concerns itself with realistic clones of real people and includes behaviour models suitable to produce virtual actors in which fluidity of movement extends as far as dynamic clothing. The project includes avatar and general content creation, distribution and integral display. The idea is that the avatar creation involves different humanoid aspects developed separately and brought together before encoding, transmission, decoding and 3D display. These aspects are the capture of a 3D model of an actor in the AvatarMe booth [65], realistic facial expressions [66], and dynamic clothing [67] to replace inadaptable texture mapping. Work also extends the booth capture technology to the capture of photo quality models of actors in a multiple camera virtual studio [68]. The time and funding factors

of the project did not allow all of these components to come together in real-time, though individual successes in this chain of events were accomplished by each of the participants.

The idea is that the integrated avatar information of each humanoid aspect is transmitted using MPEG-4 and at the final receiving end is the integral display. Unfortunately, at the time of project completion, dynamic clothing information was not fully integrated and a possibly suitable MPEG-4 distribution encoder/decoder was discovered too late for modifications and interfacing for the final demonstration. However, 3D source data for stills originating from the booth and animated avatars from the virtual studio were available for texture mapping in the integral imaging environment.

Standardized methods of representing 3D human models are provided by the VRML Humanoid Animation Working Group (H-Anim) [69] and exist as 'Protos' within the VRML2 format and can be transmitted in Binary Format for Scenes (BIFS – a part of the MPEG-4 standard) format for animation coding and streaming. The 'Protos' contain the underlying skeletal structure with 17 joints to synthesize the gross movements of the body. The avatar structure is texture mapped with the image of a 'real' person. It was necessary, therefore, to write a complex parser/translator to present the humanoid data suitable for the integral renderer (see Appendix H).

The integral renderer and the translated scene file, in turn, also had to be modified to provide for texture mapping to enable any parameter changes to this 'integral imaging avatar rendering system' the translator and modified integral renderer reads the scene file heading parameters from a separate parameter list (Figure 7.6).

The integrally rendered, texture mapped avatars (e.g. Figure 7.7) generated from the virtual studio can be animated as if they were statues i.e. fixed pose and rotating, but animation and avatar data for other avatars, although still in VRML2 format, is generated by the developers in a single segment 'Proto'. This required further minor modifications of the parser/translator that was originally designed for multiple segments. The integral animation sequence was MPEG-2 encoded and displayed

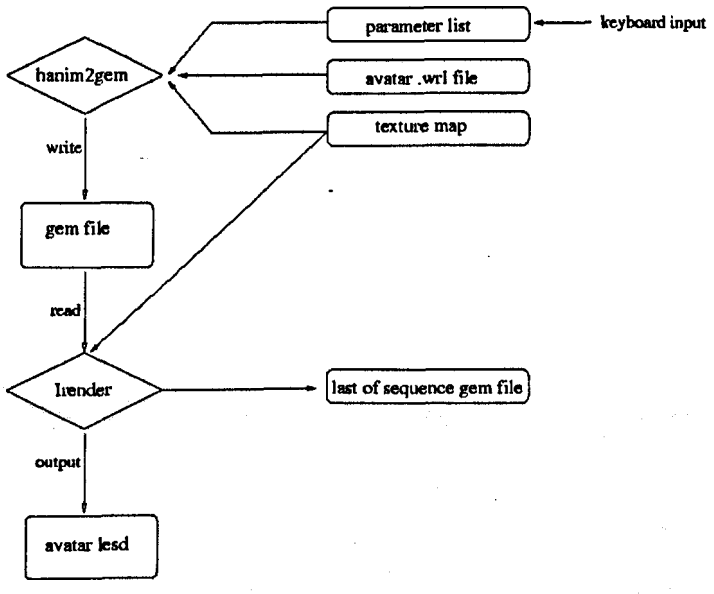

Figure 7.6: Avatar texture mapping translator and integral renderer flowchart

initially on a Samsung LCD at 96.212dpi and, hence, had limited depth resolution. Later these integral frame sequences were displayed on the high-resolution T221 LCD and this enabled more of the scene to be replayed in front of the screen (example frames can be seen in Figure 7.8). [70].

The actual flow of the animation can be more clearly seen in Figure 7.9, which is a 2D amalgamation of selected frames from different viewpoints. This helps to orientate the scene for the best production set up before integral rendering.

Tests were carried out to find the largest possible area of pixels on the 9 million-pixel T221 that real-time playback (off-line generation but played back at a reasonable frames/s rate) could be accomplished. An integral video in MPEG-2 format displayed noise on the high-resolution LCD that was not noticeable at lower resolutions. Therefore, animation frames were packed into a lossless avi format. Without hardware help only a 1024x1024 pixel area ran at a reasonable rate. The LCD pixel area is 3840x2400; hence the animation covered only 1/9th of the screen.

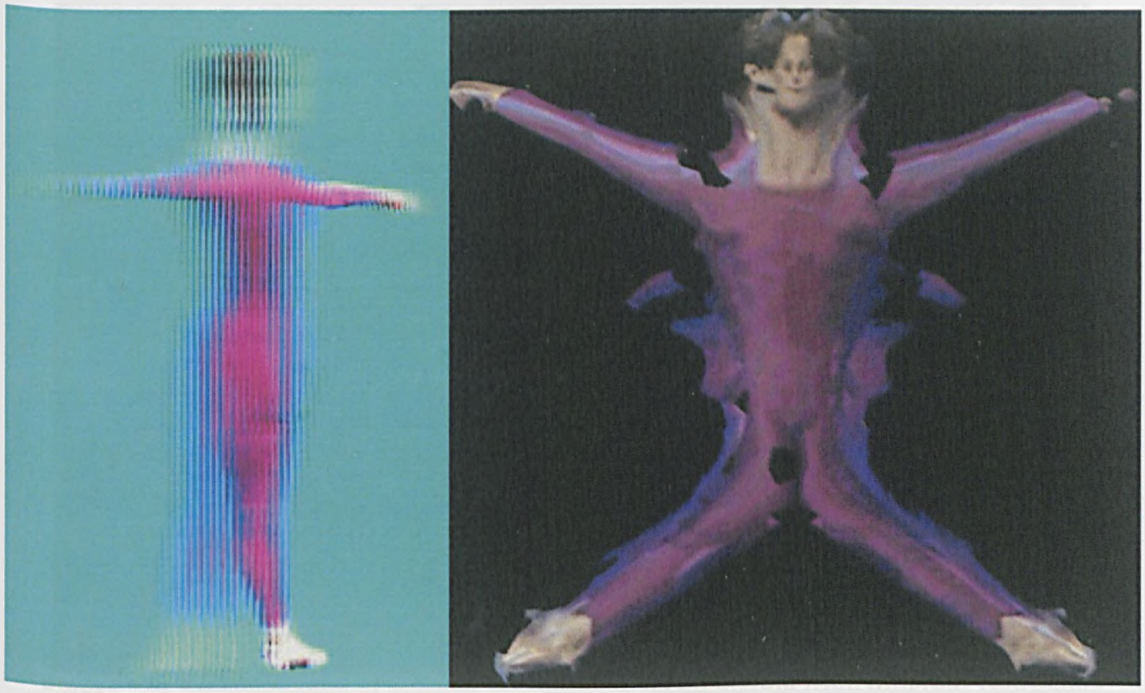

Figure 7.7: Left:'Real' ballerina avatar LeSD Right: avatar texture map

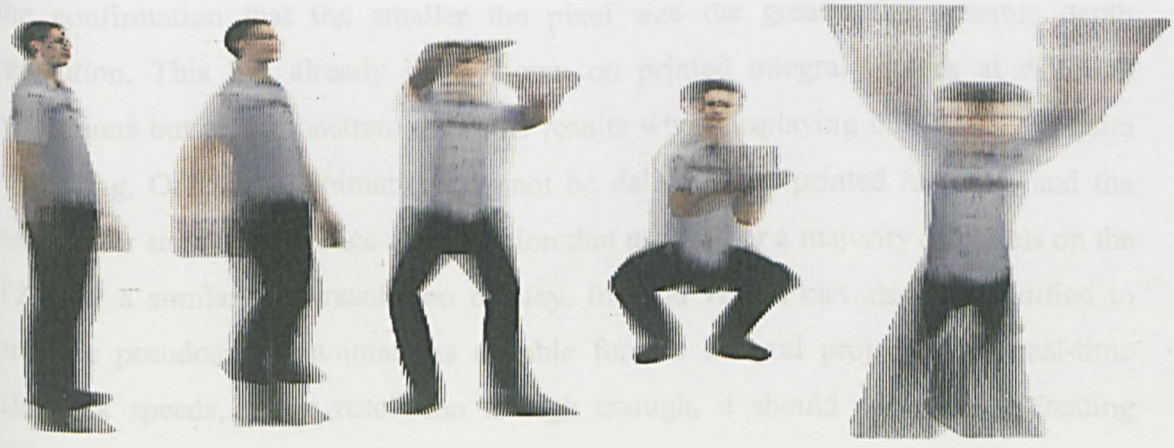

Figure 7.8: Animation frames on high-resolution display set at 50% depth

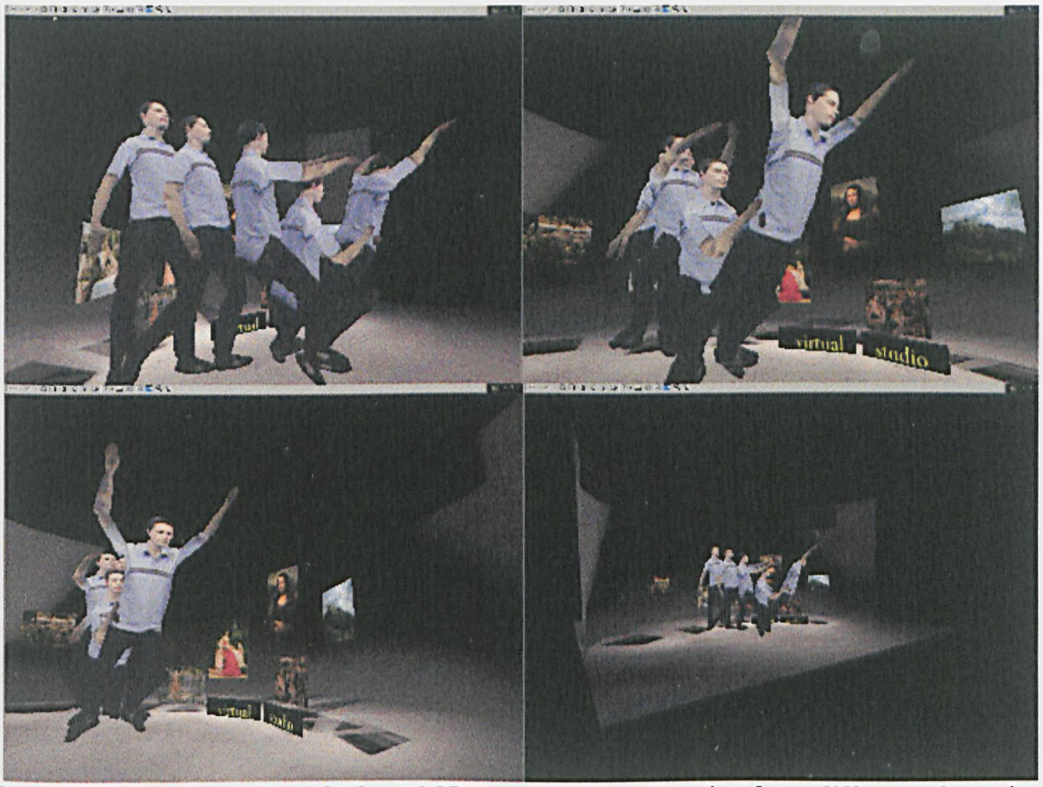

Figure 7.9: Amalgamation of selected 2D frames of an animation from different viewpoints

7.7 Conclusions

The most significant discovery during these production and display experiments was the confirmation that the smaller the pixel size the greater the possible depth resolution. This has already been shown on printed integral images at different resolutions but to demonstrate the same results when displaying on an LCD is quite rewarding. Of course, animations cannot be delivered on printed material, and the next major step is to produce an animation that uses all, or a majority of, pixels on the T221 or a similar high-resolution display. Integral videos can also be modified to produce pseudoscopic animations suitable for the integral projector. At real-time playback speeds, if the resolution is high enough, it should prove an interesting experiment.

MPEG-4 BIFS streaming of virtual environments and the corresponding animations, featuring an integral display would be a major step. The content could then be made

available to users worldwide through the Internet. However, this would remain in the domain of experts and researchers for some time due to the processing power required at the client end and the expensive high-resolution displays needed. This could be a future direction for integral imaging as a display modality. The generation of real-time integral imaging is the subject of the next chapter.

8

Real-Time Generation

8.1 Introduction

The most obvious speedup available is the use of standard graphics cards that take all of the hard work and bottlenecks out of the overworked central processing units. Cards have been developed and evolved to run faster and faster especially for forward geometric projection techniques used in the computer games market. These provide texture mapping and Gouraud shading, as well as the time-consuming triangle projection calculations. The ideal arrangement for a real-time compositing technique (Chapter 6) is to generate 2D images, from different but calculated positions of a scene simultaneously, using a PC cluster fitted with DVI outputs and compositing hardware to multiplex the outputs. This would enable the use of any standard graphics software and/or decoder with the minimum (or none) of code modification and full use of off-the-shelf graphics cards. The compositor output is ideally sent to a high-resolution display, increasing the strain on the system to reach real-time, but necessary in order to produce achievable integral video. Unfortunately this ideal set-up is not possible within this thesis. However, taking any commercial or open source 2D imaging software, generating the required N number of 2D images for multiplexing, compositing these images using software and displaying on an LCD fitted with a lens array, would be the next step and be sufficient to test the basic concepts of the approach.

8.2 Compositing sub-images in hardware for real-time processing

The differences between producing standard monoscopic computer generated video and integral video for real-time is that integral imaging requires multiple images of the scene that have to be multiplexed and composited. Another difference is that integral video must be displayed on an LCD not a standard PC monitor.

It is proposed that integral imaging is a suitable display technology for use wherever static or dynamic computer generated monoscopic images are viewed, in applications such as engineering and scientific visualisations, multimedia presentations, entertainment, web pages or shared virtual worlds. If this is achieved the user would be allowed to view these images volumetrically and perceive the electronic world in the same way as the real world. On a broader scale this also includes real scenes and 3DTV requiring a real 3D camera [71].

8.2.1 Requisites for a real-time renderer

This ideal is almost possible at the present time but requires more computing power. This can be provided by parallel processor machines or by the use of PC clusters. In addition a hardware compositor is required for pixel routing, and the ability to set the different projection points or viewpoints on each node is needed. A high resolution LCD (i.e. small pixel pitch – <0.12mm) is necessary to form high quality integral images that allow greater depths to be comfortably resolved for standard monitor display area sizes. This puts real time operation as a difficult goal as all the requisites are at the extreme end of computer power, bandwidth, and temporal requirements.

Each node must have its designated viewpoint and simultaneously receive the un-rendered scene information. The PCs must have video cards with Digital Video Interface (DVI) outputs not only for better quality video, through by-passing the digital-to-analogue and analogue-to-digital converters in PC and flat panel displays respectively, but also because DVI has high-bandwidth abilities and digital signals facilitate the routing process. For transmit and receive over the Internet if the server end meets these requirements and transmits LeSDs there may be problems with the encoder/decoder as LeSDs have unique intensity distributions. Therefore, compatibility issues may limit the systems generality. Even so, it is worth bearing in mind that at the client end the user would only need to purchase a high-resolution LCD fitted with the appropriate lens decoder. Integral imaging compression techniques have already been developed and refined at the 3D & Biomedical Imaging Group at De Montfort University [72][73][74], hence resolving the compatibility issues.

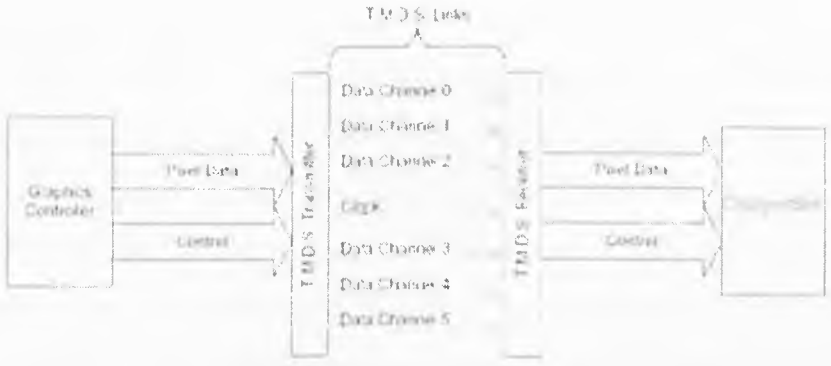

Figure 8.1: Two TMDS links

The DVI specification requires at least one transition-minimized differential signalling link (TMDS) - a serial encoding protocol used to transmit data over a DVI connection. The TMDS link consists of three data channels (RGB) and one clock control channel. According to the DVI specification [75], a TMDS link may operate at up to 165MHz and a single 10-bit TMDS link offers 1.65Gbps of bandwidth, which is enough for a 1920 x 1080 resolution refreshed at 60Hz on a digital flat panel. In order to keep the specification as flexible as possible, a second TMDS link may be used (Figure 8.1). This link must operate at the same frequency as the primary link, therefore, for example, in order to obtain 2Gbps of bandwidth each link must operate at 100MHz (i.e. 100MHz x 2 x 10).

Higher rate DVI interfaces can also be achieved using two single link DVIs. So a dual link DVI interface should support formats up to a maximum clock rate of 330Mhz. Xilinx state that the DVI double link can distribute video data at 9.9Gbps! [76]. The bandwidth requirement for a single PC to generate and display 24-bit RGB data on the IBM T221 QUXGA (3840x2400) LCD at its vertical refresh rate of 47Hz is 10.4Gbps (i.e. 3840x2400x24x47). However, to date no graphics card to drive the T221 with sufficient scan-out bandwidth at a high refresh rate is available. However, the T221 can be driven by up to *four* separate synchronised DVIs, so even higher bandwidth can be achieved.

Considering the latest generation of FPGA's from Xilinx - Virtex-II, and the current chipsets from Silicon Image and Texas Instruments (i.e. the SIL161A chip that operates as a dual link receiver) it is just about feasible to produce a high-resolution

real-time integral compositor/multiplexer. A horizontal uni-directional parallax multiplexer would not be as difficult to implement as an omni-directional one. Therefore it is sensible to attempt to achieve a horizontal parallax only system as an initial system. Vertical multiplexing would probably require a very high speed DDSDRAM store and a considerably increased development time. If the whole display presents too large a task then smaller areas on the display would suffice until real-time is reached. Care would need to be taken that automatic resizing does not occur as this would destroy the integrity of the intensity distribution. This may be accomplished by selecting a resolution that has the same aspect ratio as that of the display.

To begin production of the compositor/multiplexer a dedicated layout and fabrication cycle should be organized, then design time and debug time allowed for, and finally, a second iteration if there is a problem. Whichever system is used it is necessary to identify and be aware of the details of:

1) The graphics card: connector and electrical interface type, and the actual system clock rate for the resolution of interest.
2) The monitor type: actual connector and electrical interface type, and actual range of system clock rates - including blanking.
3) The connector and receiver types may be completely different to the details given by the LCD manufacturer that are usually given for the 'naked' panel with its low voltage differential signalling (LVDS) interfaces instead of the TMDS information.

An almost ready-made option, though dependent on research products becoming commercially available, is the Chromium system [77] that can be used to manipulate and control the streams of graphics API commands on the PC cluster. Hardware such as Lightning-2 [78], a display subsystem for such a cluster, would appear capable of handling the multiplexing. It has the advantage of *"being able to allow any pixel data generated from any node to be dynamically mapped to any location on any display"* and connects to video cards via DVI, hence it requires no modifications to accelerator hardware or device drivers.

8.2.2 Software 'proof of principle'

A software 'proof of principle', that replicates the hard-wired viewpoints on the cluster nodes and the DVI-based pixel routing network, is a valuable step in the design of the compositor as well as proof that the technique works. Similarly, it provides a working blueprint to directly implement on Chromium and Lightning-2 if they, or others like them, become available.

The idea is to be able to generate integral images using any application that produces 2D images from descriptions of 3D scenes without modifying the code of the application. Scene formats such as VRML2 or MPEG4 utilize Web technology to bring networked virtual reality to the end-user and versions of these are being improved and updated all the time, as are the applications and browsers. It will not be necessary to keep up with all of these developments, or to continually re-modify code within these applications to produce integral images using the pinhole technique (assuming that the code is not binary). *The technique is independent of the application or scene format.* Each (VRML) virtual world is a self-contained environment that is downloaded for navigation and although bandwidth directly affects the download time, once loaded the user's experience is dictated by the performance of the integral imaging system. The client viewer and the server are in intermittent contact as the user clicks on links to download and navigate other worlds. Streaming the 3D content (MPEG-4) improves the experience for the user and reduces the waiting time to become involved in the virtual scene.

Blaxxun [79] is the chosen browser for this experiment and the scene file is a VRML2 view of three eggs with letters texture mapped on their surfaces and a background of ground and sky. A LCD with a resolution of 1280x1024 is used that gives an integer number of pixels per lenslet (8) when used with a semi-cylindrical array of 12 lines per inch, the pixel pitch being 0.264mm. Calculations for the camera positions on the viewpoint plane must be performed and are dependent upon the chosen size of the aperture. In this experiment the aperture is taken as the same size as the width of the display (i.e. 1280x0.264=337.92mm). Using twice as many viewpoints (16) as pixels behind each lenslet enables anti-aliasing as described in sub-section 6.3.3.

The pinhole technique developed theoretically in Chapter 2 allowed each scene projection access to the whole lens array by eliminating the virtual barriers between lenslets. This is equivalent to orientating the angle of each projection point such that its centre of field of projection is targeted on the midpoint of the image plane. If 2D views, in this experiment, were taken from forward facing cameras then the centre of the scene would be positioned in consecutive images from the extreme left to the extreme right. Hence, orientation angle calculations for each camera along with the camera position calculations must be performed. The image plane, however, is not static, but rotates for each viewpoint to align itself at right angles to each camera's centre of field of projection. The pivot point is the centre of curvature of the central lenslet. This angle of rotation is relatively minimal and could eliminate a very small amount of perspective distortion (small due to the large distance between aperture and lens array). Due to the image plane's perpendicular alignment, with respect to each viewpoint, an alternative to calculating camera orientation angles and positions is to simply have one centrally located viewpoint and rotate the scene correctly for each successive sub-image. However, in practice, different graphics applications rotate scenes around slightly differently calculated rotation points and the scene objects in the integral images thus produced tend to be deformed.

Figure 8.2 reveals the dimensions necessary for the calculations. The value of the distance from each viewpoint through the centre of curvature to the image plane (a_n) changes for each viewpoint location (V_n). These values are the z-coordinates for each viewpoint. Similarly, the distance from the virtual lens arrays central optical axis to each viewpoint (x_{vn}) varies. These values are the x-coordinates. The angles (θ_n) are rotations around the y-axis. Hence:

$$a_n = \sqrt{x_{vn}^{\,2} + d^2} + t - r + currentZ$$

$$x_{vn} = x_{vn-1} + \frac{A}{2n-1} + currentX$$

$$\theta_n = \tan^{-1}\left(\frac{x_{vn}}{a_n}\right) + currentOrientation$$

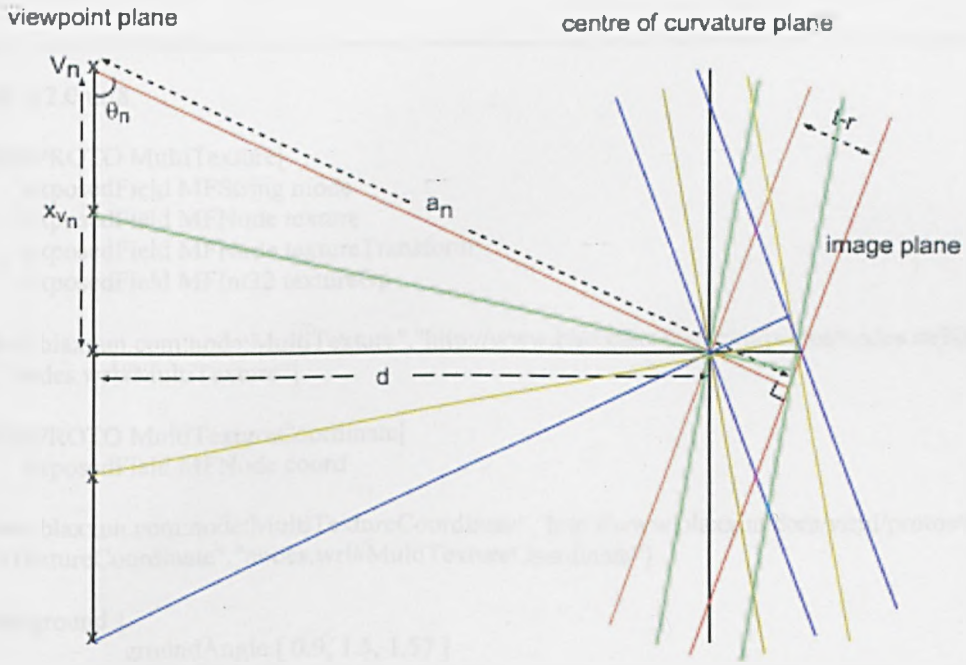

Figure 8.2: Camera orientation and position calculations

These camera viewpoint parameters are the values that would be required to be hardwired to each node in the cluster.

A program written to simplify the task outputs each camera position and orientation for any given system (Appendix I, Part 1). The user inputs are the lenslet pitch, width and height of the display, the pixel pitch and the number of viewpoints. The output is in a format that VRML2 understands and is inserted directly into the scene file e.g.

```
#VRML V2.0 utf8

EXTERNPROTO MultiTexture[
        exposedField MFString mode
        exposedField MFNode texture
        exposedField MFNode textureTransform
        exposedField MFInt32 textureOp
]
["urn:inet:blaxxun.com:node:MultiTexture","http://www.blaxxun.com/vrml/protos/nodes.wrl#MultiTe
xture","nodes.wrl#MultiTexture"]

EXTERNPROTO MultiTextureCoordinate[
        exposedField MFNode coord
]
["urn:inet:blaxxun.com:node:MultiTextureCoordinate","http://www.blaxxun.com/vrml/protos/nodes.wr
l#MultiTextureCoordinate","nodes.wrl#MultiTextureCoordinate"]

    Background {
                groundAngle [ 0.9, 1.5, 1.57 ]
                groundColor [
                                0 0.333 0,
                                0 0.4 0,
                                0 0.5 0,
                                0.62 0.67 0.60
                ]
                skyAngle [ 0.9, 1.5, 1.57 ]
                skyColor [
                                0 0 0.1
                                0.21 0.18 0.66,
                                0.2 0.44 0.85,
                                0.77 0.8 0.82
                ]
        }

# 16 VIEWS
 DEF V1 Viewpoint { position 168.96 -80 557.99527   orientation 0 1 0 0.309428 }
#DEF V2 Viewpoint { position 146.432 -80 551.59185  orientation 0 1 0 0.270291 }
#DEF V3 Viewpoint { position 123.904 -80 546.0434426 orientation 0 1 0 0.230286 }
#DEF V4 Viewpoint { position 101.376 -80 541.3763325 orientation 0 1 0 0.189516 }
#DEF V5 Viewpoint { position 78.848 -80 537.6134737  orientation 0 1 0 0.148100 }
#DEF V6 Viewpoint { position 56.32 -80 534.7739545   orientation 0 1 0 0.106165 }
#DEF V7 Viewpoint { position 33.792 -80 532.8725357  orientation 0 1 0 0.063853 }
#DEF V8 Viewpoint { position 11.264 -80 531.9192774  orientation 0 1 0 0.021310 }
#DEF V9 Viewpoint { position -11.264 -80 531.9192774 orientation 0 1 0 -0.021310 }
#DEF V10 Viewpoint { position -33.792 -80 532.8725357 orientation 0 1 0 -0.063853 }
#DEF V11 Viewpoint { position -56.32 -80 534.7739545 orientation 0 1 0 -0.106165 }
#DEF V12 Viewpoint { position -78.848 -80 537.6134737 orientation 0 1 0 -0.148100 }
#DEF V13 Viewpoint { position -101.376 -80 541.3763325 orientation 0 1 0 -0.189516 }
#DEF V14 Viewpoint { position -123.904 -80 546.0434426 orientation 0 1 0 -0.230286 }
#DEF V15 Viewpoint { position -146.432 -80 551.59185  orientation 0 1 0 -0.270291 }
#DEF V16 Viewpoint { position -168.96 -80 557.99527   orientation 0 1 0 -0.309428 }
```

The browser renders each viewpoint in turn by eliminating the current viewpoint hash. Pre-imaging positioning and appearance of the scene can be accomplished by altering values in the 'Transform' and 'Shape' nodes e.g.

```
DEF ROOT  Transform {
        rotation 0 0 0 0.785398
        translation 0.0 -41.52 80.0
        scale 1 1.84 1
        children [

 DEF ROOT-SHAPE   Shape {
         appearance Appearance {
          material Material {
             #diffuseColor 0.5373  0.1961 0.1961
              ambientIntensity 0.1033
              specularColor 1 1 1
              shininess 0.1 # 0.2875
               transparency 0.1
       }

..... rest of scene file
```

The resultant sub-images are ready for compositing (Figure 8.3).

A program was written (Appendix I, Part 2) to read in each sub-image and route its relevant pixel values to the correct positions in the composite image as derived in section 6.3. Care must be taken to read in the images in the correct order i.e. right to left if the camera panned left to right during sub-image capture; failure to do this results in a pseudoscopic integral image. The composite is an anti-aliased integral image with a resolution of 1280x1024 (Figure 8.4).

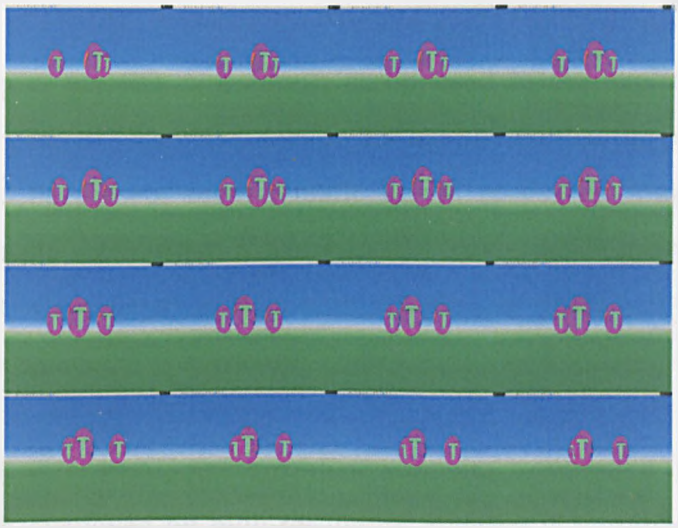

Figure 8.3: 16 sub-image views taken from camera panning left to right with readjusted camera angle and distance from the scene for each shot

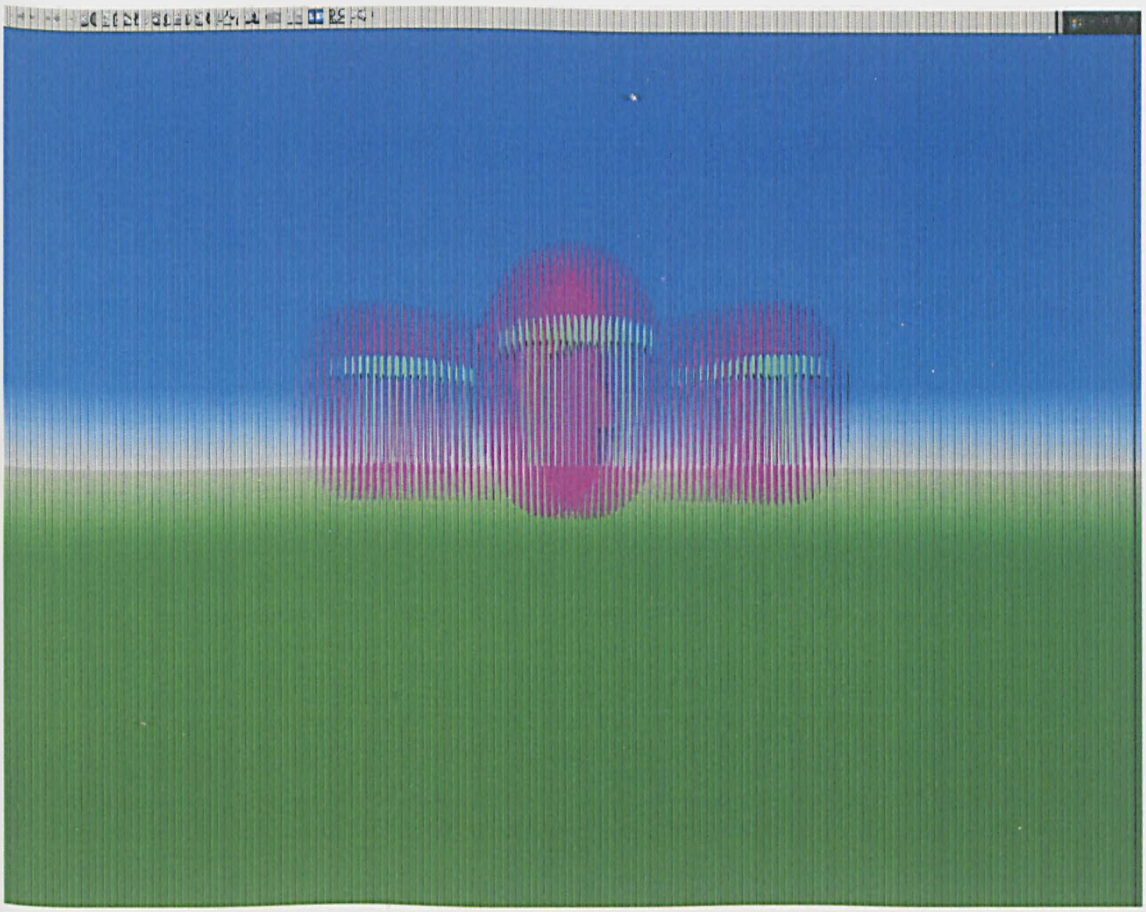

Figure 8.4: Anti-aliased LeSD generated by compositing multiple 2D sub-images originating from a VRML2 scene file and rendered on Blaxxun

8.3 Conclusions

Although the inputs are 2D projections, as opposed to the explicit 3D finite-sized aperture model inputs of Chapter 5, section 2.4 has shown that the pixel intensities are those that would be present for a pinhole model producing integral images. This concludes that in the method put forward in this thesis, explicit depth information is the same as that when depth is implicitly encoded as positional disparity between different projections.

The benefits of this technique are that computer generated real-time integral (streaming) video displayed on high-resolution displays is possible with present day technology. There is no necessity for code modification of the graphics engine and only a limited incursion into the scene file. Full use can be made of video cards with DVI outputs, and digital-to-analogue and analogue-to-digital converters are bypassed giving more speed and higher image quality.

9

Conclusions

The development of this research, from the initial exploratory computer generated spot and mesh images to two fully integral, rendering and animation resources in both finite-sized aperture and pinhole models, has shed light on this previously unexplored territory. Discussions and conclusions pertinent to each of these progressive studies have been included in the respective chapters and here some specific and general conclusions of the work are stated or reiterated.

When the initial mesh integral images produced the first signs of the magic of autostereoscopy it provided the impetus to complete the goal of computer generated rendered integral images. However, a major stumbling block to the first part of the research was that the optical and dimensional parameters of the available microlens arrays were not known to any degree of useful accuracy. For images that did not quite work as expected, it was not known whether or not it was due to mistakes in the code, a flaw in the design, or incorrect parameters of the virtual lens array that must match those of the real lens array. During this part of the work ray traced integral image production had already achieved some success but integral artefacts still presented similar problems. It was not until a relatively cheap source of good lenticular sheets with known parameters was found that these problems were eliminated and the resultant images reacted in expected or resolvable ways to most code changes. It was then possible to track down problems as they occurred knowing that the parameters were correct and this allowed the fine-tuning of the images.

Integral imaging is an ideal candidate for an all-round viewable autostereoscopic system and yet research and hence literature related to integral imaging is very limited. Research into integral imaging has been ongoing for many years by the 3D and Biomedical Imaging Group at DMU, interest, however, has started to be shown by some major research groups at NHK, Seoul University and the BBC.

Although computer generated ray traced integral images have been developed at DMU there is an obvious need for faster generation of integral images especially due

to the computer intensive nature of their production. It is shown in this thesis, by derived equations, when analysing a simplified diagrammatic light structure of pixels replaying through a microlens array, that the light mesh is orthogonal and static and therefore not an acceptable blueprint to implicitly follow in the design for the computer generation of integral images. A new model is required that has perspective. Methods have been put forward to accomplish this, these methods being 3D-from-2D (pinhole) model and the full lens array (finite-sized aperture) model. Both enable perspective integral images to be produced essentially by allowing each projection point on the aperture access to the whole lens array. This, in turn, requires that no opaque barrier should exist between the lenslets of an array allowing projected rays to intersect pixels through an adjacent lenslet to that of the 'parent' lenslet of that pixel, if required. The consequence of this, for both methods, are that only a relatively few scene projections are required to fulfil the definition of an integral image. This brings the possibility of real-time integral imaging much closer to a reality than was previously expected. The methods put forward not only allow perspective integral images to be produced but also variable perspective and a variable geometry to be replayed even though the replay geometry of a microlens array itself is static. This involves moving the aperture (i.e. projection point plane) to any required position to change the perspective nature of the integral objects and the technique is compatible with the camera in any standard computer graphics program. The derived equations show that the methodology is sound and the integral images produced will not display double imaging. The equations can be converted into suitable code and assimilated by the basic integral imaging ray trace program allowing *it* to generate *perspective* integral images.

A number of important features have come to light during exploratory integral imaging experiments that produce integral mesh images:

- The projection of the perimeters of object triangles is necessary not only for mesh programs but also for a finite-sized aperture, rendering model. Therefore the building of a concise 3D line-drawing algorithm is required to generate fixed, equally spaced points, independent of the perimeter line lengths, and at a calculated optimum spacing suitable for integral graphics.

- Back-face culling is an action to be performed in any 3D graphics program due to the amount of processing that is saved but for integral imaging it is not performed with respect to a single viewpoint, as in standard graphics, but for a whole aperture.
- The effects of diffraction are so small that they are not a concern for the integral capture models.
- A finite-sized aperture model can be used when a non-integer number of pixels per lenslet are present.

Knowledge of these features was helpful during the development of the finite-sized aperture models.

Any standard rendering program requires options such as the different shading algorithms and different projection modes but integral imaging also has options due to the different structures of microlens arrays. Possible types of array are semi-cylindrical, circular and packed hexagonal lenslet arrays. The rendering models developed and described in this thesis allow many modes of operation based on these possibilities.

An important feature when using optical arrays has come to light. This is to ascertain the system matrix and provide the positions of the unit and focal planes and hence arrive at a clear understanding of the characteristics of the optics. Therefore, a full optical matrix examination of each lens array is required to be carried out for real-camera work and in the computer generation of integral images. This has proved beneficial in the creation of integral images by showing that the rear focal plane of a plano-convex lenslet within an array is not measured by using the rear focal length of the lenslet. The rear focal length is the distance between the rear unit plane and the rear focal plane but the rear unit plane, in the example of section 5.5, is almost coincident with the centre of curvature. This displacement of the unit plane from the vertex of a lenslets curved surface is quite considerable (1.03mm in the example) and without the analysis a finely tuned approach would not be successful.

Due to the complexities of computer generated integral capture, program flow considerations have been analysed to provide fast and efficient methodologies in the production of integral images. Two distinct variations of the finite-sized aperture model have been developed in this research. The first and slowest is the most accurate with respect to colour definition and easily accommodates anti-aliasing. It allows additive pixel intensities to increase naturally but requires more storage space in the form of a compositing buffer to add and store the distribution result of each successive scene projection. The second variation uses single hit intensities whereby the program flow allows faster image processing and less storage space is required, though colour replication is less accurate. Anti-aliasing, for this variation, requires post-processing and unwanted transparency artefacts tend to be present.

Though the finite-sized aperture models produce integral images much faster than the integral ray tracing technique (at the time of writing) and off-line animations are quickly and easily produced, a processing rate of at least 15 frames a second for real-time is well beyond their capability. However, a second method has real-time potential. This method, the forward projection pinhole model, directly maps object points to pixels by an understanding of the structure of the LeSD when lenslets behave as pinholes. It has been shown that it is possible to generate integral images by extracting pixel intensity information from captured 2D images of a scene from different viewpoint positions equally aligned on a plane. This pixel intensity information is the same as that produced if the virtual lens array was present and the lenslets modelled as pinholes. These images do not suffer from "flipping" effects that occur when discretised planes are present because the method enables all-round seamless viewing which is a hallmark of integral imaging. The coding of the finite-sized aperture method has been metamorphosed into pinhole model code with 'on-the-fly' filtering and multiplexing to speed up the processing. Two forms of anti-aliasing are possible, one of which is implemented into the program and allows additive pixel intensities to develop by increasing the frequency of projection locations.

Although during the course of this research many computer generated integral images have been produced and analysed for each method and variation of each method that produced them, a formal approach to stretch and test their full capabilities has been necessary. The first of these came in the form of 'graphics science meets art' to

produce integral cyber-structure animations in collaboration with a cyber-sculptor. It required the production of many intermediate programs to be written; to translate a 3DS Max produced VRML2 file to integral format, to prepare the initial state of the scene and produce correct scaling factors for an 800 frame, anti-aliased, off-line animation, the writing of script files and performing various minimal modifications to the integral program used. A resulting practical understanding of the effects of pixel size became apparent when displaying both the animation and also static pseudoscopic integral images produced for an integral projection system. All were generated for high and low resolution displays but the high-resolution LCD displays were resoundingly superior in their capability to generate holographic effects (displaying parts of the scene in front of the decoding microlens array) and large volumetric integral projections were possible using high resolution transparency prints.

The Prometheus project initially required that texture mapping be incorporated into the integral programs. The texture mapping is primarily directed at the production of integral avatars, the original avatars being either generated in an avatar booth or in a virtual studio. These are avatars of actual people that are represented in VRML2 format in a special 'Proto' designed by H-Anim. It was necessary to write file translation programs and modify the integral scene file format and integral imaging programs and generate high and low resolution integral animations. It was not possible to generate the high-resolution animations for a QUXGA display (IBM T221 LCD) in real-time playback for the whole display area but a part of the total display area was used providing a more limited success.

The future of forward projection integral imaging, using interpolative shading techniques, is the real-time generation of these images. The pinhole method (3D-from-2D method) can achieve this aim with today's technology and would enable any standard 3D graphics program to be used to generate the images integrally. This would not necessitate the use of any code modifications but would require a PC cluster, DVI interfaces, and either a purpose built multiplexer/compositor or certain research products becoming available.

The research contained in this thesis has produced integral images using the faster interpolative shading methods of standard computer graphics. At the beginning of the research this is all that was to be attempted and hopefully accomplished. Much more has been accomplished, though, not only in the understanding of the structure of a LeSD, but also in the practical production of computer generated integral images. From simple spot and mesh images to fully rendered off-line animations and animated realistic looking avatars depicting real people on a volumetric display system. It is a viable way forward and more computer graphics related conferences are accepting papers related to this and similar concepts. The subject is usually seen as being on the fringe of things, like holography or stereo films requiring glasses but more people and organisations are now seeing that as monochrome TV lead to colour TV, then its time for a change, for another dimension – for 3DPC and 3DTV.

Appendix A (see 1.5)

Design and construction of a 3D integral imaging camera using two-tier micro optics

Appendix B (see 2.4)

Pixel hit calculation code

```
/* Program to calculate pixel hit as in Chapter 2. Can have ka changing with kl fixed or vice-versa
   Outputs to screen and to text file proof1.txt        */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

main()
{
 double V,Y,A,a,b,kp1,kp2,P,m;
 int ka=0,N,n,x,kl=0;

 FILE *fopen(),*fv;
 fv = fopen ("proof1.txt", "w");

/*************************************************************************************************************
   MAKE CHOICES HERE  */

kl=1;/* ka=1; */        /* kl lenslet number - or ka projection point number */
N=226;                  /* number of lenslets */
P=2.1167;               /* pitch of lenslets */
n=17;                   /* number of pixels per lenslet */
x=1;                    /* distance multiplier of aperture to lens array. x=1 - aperture same size as lens array at distance d
                           x=2 - aperture 2d from lens array (hence twice as big) etc */
*************************************************************************************************************/

fprintf(fv,"\n Program to calculate pixel hit as in Chapter 2. Can have ka changing with kl fixed or vice-versa\n");
fprintf(fv,"\nb is the distance in pixels from centre pixel behind relevant lenslet to hit on image plane");
fprintf(fv,"\nkp1 is the pixel number of the centre pixel behind the relevant lenslet");
fprintf(fv,"\nkp2 is the final pixel number of the hit\n\n");

A=(N*P); m=(n-1)/2.0;

fprintf(fv,"\n kl=%d ka=%d N=%d P=%f n=%d A=%f x=%d",kl,ka,N,P,n,A,x);

if (kl==0) {
          printf("\nProjection Point Number = %d  x = %d\n",ka,x); }
          fprintf(fv,"\nProjection Point Number = %d  x = %d\n",ka,x);
          if (ka==0) { printf("\nLenslet Number = %d   x = %d\n",kl,x);
            fprintf(fv,"\nLenslet Number = %d   x = %d\n",kl,x); }
          if (kl==0) { printf("\n Lens No.  V     Y    a      b     kp1     kp2     pixel span behind lenslet\n");
            fprintf(fv,"\n Lens No.  V     Y     a     b     kp1     kp2 pixel span behind lenslet\n"); }
          if (ka==0) { printf("\n Proj. Pt. No. V   Y    a     b     kp1     kp2     pixel span behind lenslet\n");
            fprintf(fv,"\n Proj. Pt. No.  V     Y  a      b      kp1     kp2     pixel span behind lenslet\n");
        }

for (ka=1;ka<=N;ka+=1) /* or change to for(kl=1;kl<=N */
        {
          V=((A*x)*((2*ka)-1))/(2*N);
          Y=((A*(x-1))+(P*((2*kl)-1)))/2.0;
          a=(x*A)-V-Y;
          b=-(a*n)/(x*A);
          kp1=-(n*(N-(2*kl)+1))/2.0;
          kp2=-(((n*(N-(2*kl)+1))/2.0)-b);
          printf("\n %d     %.2f  %.2f  %.2f  %.2f  %.2f   %.2f    %.2f  %.2f",ka,V,Y,a,b,kp1,kp2,(kp1+m),(kp1-m));
          fprintf(fv,"\n %d       %.2f  %.2f  %.2f  %.2f  %.2f   %.2f      %.2f  %.2f",ka,V,Y,a,b,kp1,kp2,(kp1+m),(kp1-m));
        }
printf("\n\n");
}
```

Appendix C, Part 1 (see 3.3.1)

3D line drawing – dependent on start/end coordinates

3D line drawing – distances between points making up lines of triangle perimeters are dependent on the start and end point coordinate values.

```
void lines_step (vector *v1,vector *v2,vector *v3,vector *lines,int *points)
  {
   float alpha,beta,delta,a[3],b[3],c[3],tempa,tempb,tempc;
   int n,u,i=0;

   points[0]=points[1]=points[2]=STEP;

   a[0]=v1->x; b[0]=v1->y; c[0]=v1->z;  /* Storing triangle corner coordinates one triangle at a time*/
   a[1]=v2->x; b[1]=v2->y; c[1]=v2->z;
   a[2]=v3->x; b[2]=v3->y; c[2]=v3->z;

   tempa=a[0];tempb=b[0];tempc=c[0];  /* temporarily saving 1st corner info */

   for (u=0;u<=2;u=u+1)
     {
      if(u==2)
       {
         alpha=a[u]-tempa;        /* calculating lengths of each perimeter line in each axis direction */
         beta=b[u]-tempb;
         delta=c[u]-tempc;
       }
      else
       {
         alpha=a[u]-a[u+1];
         beta=b[u]-b[u+1];
         delta=c[u]-c[u+1];
       }
      if (alpha!=0) alpha=alpha/STEP;       /* STEP is no. of points in each line, set by user */
      if (beta!=0) beta=beta/STEP;   /* alpha, beta and delta now become the spaces between points*/
      if (delta!=0) delta=delta/STEP;

      for (n=0;n<STEP;n++)
        {
         if (alpha!=0) a[u]=a[u]-alpha; /* spaces are incrementally added on to start points */
         if (beta!=0) b[u]=b[u]-beta;
         if (delta!=0) c[u]=c[u]-delta;
         lines[i].x=a[u]; lines[i].y=b[u]; lines[i++].z=c[u];  /* each point is stored in lines[] */
        }
     }
}
```

Appendix C, Part 2 (see 3.3.2)

3D line drawing – independent of start/end coordinates, not concise

3D line drawing – distances between points making up lines of triangle perimeters are the same irrespective of the start and end point coordinates. The code, however, is not very concise.

```
int lines_space(vector *v1,vector *v2,vector *v3,vector *lines,int *points)
 {
 float alpha,beta,delta,a[3],b[3],c[3],tempa,tempb,tempc;
 float non_planar_diag,planar_diag,lambda1,lambda2,dx,dy,dz,n;
 int u,ta,tb,td,sum,i=0;

 a[0]=v1->x; b[0]=v1->y; c[0]=v1->z;        /* As Appendix C Part 1 */
 a[1]=v2->x; b[1]=v2->y; c[1]=v2->z;
 a[2]=v3->x; b[2]=v3->y; c[2]=v3->z;
tempa=a[0];tempb=b[0];tempc=c[0];
for(u=0;u<=2;u=u+1)
  {
    if(u==2)
     {
          alpha=a[u]-tempa;          /* note: alpha, beta and delta are lengths of lines not angles */
          beta=b[u]-tempb;
          delta=c[u]-tempc;
     }
    else
     {
          alpha=a[u]-a[u+1];
          beta=b[u]-b[u+1];
          delta=c[u]-c[u+1];
     }
        dx=0;dy=0;dz=0; ta=0;tb=0;td=0;              /* Initialising */
    planar_diag=sqrt((alpha*alpha)+(delta*delta));   /* see Fig 3.5 a */
    non_planar_diag=sqrt((alpha*alpha)+(beta*beta)+(delta*delta)); /* length of line in question */
    points[u]=floor(non_planar_diag/space);          /*counting no. of points for each perimeter line*/
    if (alpha!=0) ta=4; if (beta!=0) tb=2; if (delta!=0) td=1; /* orientation possibilities */
    sum=ta+tb+td;
    if (sum==7) { lambda2=atan(delta/alpha); lambda1=atan(planar_diag/beta); }
    if (sum==3) {   if ((delta==0)||(beta==0)) lambda1=0;
                          else lambda1=atan(delta/beta);     }
    if (sum==6) {   if ((alpha==0)||(beta==0)) lambda1=0;
                          else lambda1=atan(alpha/beta);     }
    if (sum==5) lambda2=atan(delta/alpha);

    for (n=space;n<=non_planar_diag;n=n+space)  /* same space iterations */
       {
        if (sum==7) { dx=n*cos(lambda2)*sin(lambda1); dy=n*cos(lambda1);
                                  dz=n*sin(lambda2)*sin(lambda1); }
        if (sum==1) dz=n; if (sum==2) dy=n; if (sum==4) dx=n;
        if (sum==3) { dy=n*cos(lambda1); dz=n*sin(lambda1); }
        if (sum==5) { dx=n*cos(lambda2); dz=n*sin(lambda2); }
        if (sum==6) { dx=n*sin(lambda1); dy=n*cos(lambda1); }

        if (dy<0) dy=-dy; if (dx<0) dx=-dx; if (dz<0) dz=-dz;
        if (beta<0) dy=-dy; if (alpha<0) dx=-dx; if (delta<0) dz=-dz;

        lines[i].x=a[u]-dx;  lines[i].y=b[u]-dy;  lines[i++].z=c[u]-dz; /* final point coorinates store */;
       }

  }
return(i);                  /* return total no. of points */
}
```

Appendix C, Part 3 (see 3.3.2)

3D line drawing – independent of start/end coordinates, concise

3D line drawing – distances between points making up lines of triangle perimeters are the same irrespective of the start and end point coordinates. The code uses direction cosines and is more elegant than the previous code in Appendix C, Part 2.

```
int lines_dir_cosines(vector *v1,vector *v2,vector *v3,vector *lines,int *points)
  {
   float alpha,beta,delta,dalpha,dbeta,ddelta,a[3],b[3],c[3];
   float tempa,tempb,tempc,M,N,L,dir_line_seg;
   int n,u,i=0;


   a[0]=v1->x; b[0]=v1->y; c[0]=v1->z;
   a[1]=v2->x; b[1]=v2->y; c[1]=v2->z;
   a[2]=v3->x; b[2]=v3->y; c[2]=v3->z;

   tempa=a[0];tempb=b[0];tempc=c[0];

  for (u=0;u<=2;u=u+1)
   {
   if (u==2)
    {
         alpha=a[u]-tempa;
         beta=b[u]-tempb;
         delta=c[u]-tempc;
    }
   else
       {
         alpha=a[u]-a[u+1];
         beta=b[u]-b[u+1];
         delta=c[u]-c[u+1];
       }
    dir_line_seg=sqrt((alpha*alpha)+(beta*beta)+(delta*delta)); /* perim line in question */
    if (dir_line_seg==0.0) dir_line_seg=0.000001;              /* to alleviate floating pt. errors */
    L=alpha/dir_line_seg; M=beta/dir_line_seg; N=delta/dir_line_seg;          /* dir. cosines */
    points[u]=floor(dir_line_seg/DIST)+1;
    dalpha=DIST*L; dbeta=DIST*M; ddelta=DIST*N;    /* DIST=space between points */

    for(n=0;n<(points[u]-1);n++)
       {
         a[u]=a[u]-dalpha;
         b[u]=b[u]-dbeta;
         c[u]=c[u]-ddelta;
         lines[i].x=a[u]; lines[i].y=b[u]; lines[i++].z=c[u];
       }
    if (u<2)
       {
         a[u]=a[u+1];
         b[u]=b[u+1];
         c[u]=c[u+1];
         lines[i].x=a[u]; lines[i].y=b[u]; lines[i++].z=c[u];
       }
    if (u==2)
       {
         lines[i].x=v1->x; lines[i].y=v1->y; lines[i++].z=v1->z;
       }
  }
return(i);
}
```

Appendix D (see 3.4.2)

Omni-directional parallax pinhole mesh model – transfer and refraction at the vertex of each lenslet to give final pixel coordinates

Omni-directional parallax pinhole mesh model – transfer and refraction at the vertex of each lenslet to give final pixel coordinates

```
sc=dpi/25.4;

if(Zo<Zv)                    /* for points in front of the array */
 {
    reach1=((Zv-Zo)*(Yo+(A*0.5)))/(Zo-Zap);          /* area of pinholes for given point */
    reach2=((Zv-Zo)*((A*0.5)-Yo))/(Zo-Zap);
    reach5=((Zv-Zo)*(Xo+(A*0.5)))/(Zo-Zap);
    reach6=((Zv-Zo)*((A*0.5)-Xo))/(Zo-Zap);

    start_vert=floor(((A*0.5)+(Yo-reach2))/P)+0.5)*P;     /* starting pinhole position */
    start_horiz=floor(((A*0.5)+(Xo-reach6))/P)+0.5)*P;

    for (k = start_vert; k<= (Yo+reach1); k = k + P)      /* incrementing by lenslet pitch */
       for (j = start_horiz; j<= (Xo+reach5); j = j + P)  /* until end pinhole position */
        {
          Yv = k;
          Xv = j;
          flag = 1;
          if (Yo == Yv)
               y_fin = Yo;
          else
            {
              theta1=atan((Yv-Yo)/(Zv-Zo));              /* Snell's law - refraction*/
              theta2=asin((n1*sin(theta1))/n2);
              yr = f*tan(theta2);
              y_fin = Yv + yr;                  /* position hit at image plane – floating */
            }
          if (Xo == Xv)
            z_fin = Xo;
          else
            {
              theta1=atan((Xv-Xo)/(Zv-Zo));
              theta2=asin((n1*sin(theta1))/n2);
              xr = f*tan(theta2);
              x_fin = Xv + xr;
            }
         if (flag==1)
            {
              yf=ceil((y_fin+(P*0.5)+(A*0.5))*sc);      /* integer pixel values */
              xf=ceil((z_fin+(P*0.5)+(A*0.5))*sc);
            }
        }
 }

if(Zo>Zv)                    /* for points behind the array */
  {
    reach3=((Zo-Zv)*((A*0.5)-Yo))/(Zo-Zap);
    reach4=((Zo-Zv)*(Yo+(A*0.5)))/(Zo-Zap);
    reach7=((Zo-Zv)*((A*0.5)-Xo))/(Zo-Zap);
    reach8=((Zo-Zv)*(Xo+(A*0.5)))/(Zo-Zap);

    start_vert=floor(((A*0.5)+(Yo-reach4))/P)+0.5)*P;
    start_horiz=floor(((A*0.5)+(zi-reach8))/P)+0.5)*P;
```

```
for (p = start_vert; p<= (Yo+reach3); p = p + P)
        for (m = start_horiz; m<= (Xo+reach7); m = m + P)
          {
            Yv = p;
            Xv = m;
            flag = 1;
           if (Yo == Yv)
              y_fin = Yo;
            else
              {

                 theta1=atan((Yv-Yo)/(Zo-Zv));
                 theta2=asin((n1*sin(theta1))/n2);
                 yr = f*tan(theta2);
                 y_fin = Yv - yr;
              }

          if (Xo == Xv)
             z_fin = Xo;
          else
            {
                 theta1=atan((Xv-Xo)/(Zo-Zv));
                 theta2=asin((n1*sin(theta1))/n2);
                 xr = f*tan(theta2);
                 x_fin = Xv - xr;
            }
         if (flag==1)
             {
               yf=ceil((y_fin+(P*0.5)+(A*0.5))*sc);        /* pixel coordinates */
               xf=ceil((x_fin+(P*0.5)+(A*0.5))*sc);
             }
        }
}
```

Appendix E (see 3.4.2)

Hexagonal lens arrays modelled by a pinhole technique

Hexagonal lens arrays modelled by a pinhole technique - two pitches are required as the distance of one pinhole to an adjacent pinhole on the same row is shorter than that to a pinhole in the next row of lenses.

```
sc=dpi/25.4;

if (Zo<Zv)                                          /* for points in front of the array */
 {
    reach1=((Zv-Zo)*(Yo+(A*0.5)))/(Zo-Zap);
    reach2=((Zv-Zo)*((A*0.5)-Yo))/(Zo-Zap);
    reach5=((Zv-Zo)*(Xo+(A*0.5)))/(Zo-Zap);
    reach6=((Zv-Zo)*((A*0.5)-Xo))/(Zo-Zap);

    q=-1;
   for (k = -(3*P1); k<= (yi+reach1); k = k + (P1*0.5))        /*  P1 = hexa horizontal pitch */
      {
         q++;
         if (q==1)
            {
              mstart_horiz=-(1.5*P2); q=-1;                  /*  P2 = hexa vertical pitch */
            }
         else mstart_horiz=-(2*P2);
         for (j = mstart_horiz; j<= (Xo+reach5); j = j + P2)
           {
            if ((k>=(Yo-reach2))&&(j>=(Xo-reach6)))
             {
                Yv = k;
                Xv = j;
                if (Yo == Yv)
                    y_fin = Yo;
             }
             else
              {
                 theta1=atan((Yv-Yo)/(Zv-Zo));
                 theta2=asin((n1*sin(theta1))/n2);
                 yr = f*tan(theta2);
                 y_fin = Yv + yr;
              }
             if (Xo == Xv)
                z_fin = Xo;
             else
              {
                 theta1=atan((Xv-Xo)/(Zv-Zo));
                 theta2=asin((n1*sin(theta1))/n2);
                 xr = f*tan(theta2);
                 x_fin = Xv + xr;
              }

         yf=floor(((y_fin+P1)*sc)+0.5);
         xf=floor(((x_fin+P2)*sc)+0.5);

         yf=yf+vert_shift;
         xf=xf+horiz_shift;

          }
      }
}
```

```
If (Zo>Zv)                                          /* for points behind the array */
  {
    reach3=((Zo-Zv)*((A*0.5)-Yo))/(Zo-Zap);
    reach4=((Zo-Zv)*(Yo+(A*0.5)))/(Zo-Zap);
    reach7=((Zo-Zv)*((A*0.5)-Xo))/(Zo-Zap);
    reach8=((Zo-Zv)*(Xo+(A*0.5)))/(Zo-Zap);
    q=-1;
   for (p = -(3*P1); p<= (Yo+reach3); p = p + (P1*0.5))
     {
       q++;
       if (q==1)
         {
           mstart_horiz=-(1.5*P2); q=-1;
         }
       else mstart_horiz=-(2*P2);
       for (m = mstart_horiz; m<= (Xo+reach7); m = m + P2)
           {
              if ((p>=(Yo-reach4))&&(m>=(Xo-reach8)))
                 {
                   Yv = p;
                   Xv = m;
                   if (Yo == Yv)
                   y_fin = Yo;
                 }
              else
                 {

                    theta1=atan((Yv-Yo)/(Zo-Zv));
                    theta2=asin((n1*sin(theta1))/n2);
                    yr = f*tan(theta2);
                    y_fin = Yv - yr;
                 }

               if (Xo == Xv)
                     x_fin = Xo;
               else
                  {
                    theta1=atan((Xv-Xo)/(Zo-Zv));
                    theta2=asin((n1*sin(theta1))/n2);
                    xr = f*tan(theta2);
                    x_fin = Xv - zr;
                  }
               yf=floor(((y_fin+P1)*sc)+0.5);
               xf=floor(((z_fin+P2)*sc)+0.5);

/* vert_shift and horiz_shift places scene on display at required position */

                yf=yf+vert_shift;
               xf=xf+horiz_shift;
           }
     }
  }
```

Appendix F (see 4.5)

Calculation of the number of hits each pixel receives through one lens from one point

Calculation of the number of hits each pixel receives through one lens from one point
- follows the curve of the lens exactly and is more concise
Zo<Zv Finds the equivalent position of point sampled by lens of vertex zero

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#define arc   15 /* Use integer values:  actual number of rays = (arc*2)+1   */
#define num_of_lenses 30
float z_chord,y_chord,sag,Zv,Yv,P,Zo,Yo,radius,c_to_chord,f,pc,n2,thet3,sc,pt;
float sigma,thet1,thet2,yr,chord_angle,y_fin,arc_angle,yrm,pq;
int nnn,yf,equiv_Yv,num,z,b,temp1,p,t,i=1;
int pix[2*arc];
int main()
{
 FILE *fopen(),*fv;
 fv = fopen ("equiv.txt", "w");

 fprintf(fv,"\n  An object point is traced, using many rays/lens, through one lens at a time with the
                 same angle");
 fprintf(fv,"\n  and distance to each lens, hence moving the point up the vertical aZos perpendicular to
                 the same");
 fprintf(fv,"\n  value of x-aZos. The pixel hits calculated were then normalised around zero. i.e. zero is
                 now the");
 fprintf(fv,"\n  pixel level with the vertex of that lens. The results show something akin to a cyclic
                phase shift");
 fprintf(fv,"\n  for non-integer no. of pixels/lens and homogeneous for integer no. of pixels/lens\n\n");

Zv=0.0; /* P=0.5992;*/ P=1.27;
Zo=-0.5; Yo=-P; /*f=2.65; n2=1.52;*/ f=3.075; n2=1.52;
sc=11.811024;                                                    /* 25.4/dpi */
fprintf(fv,"rays/lens = %d  no. of lenses = %d  p = %.2f  f =%.2f  n2 = %.2f Zv=%.2f Zo=%.2f
angle=0\n\n",(arc*2)+1,num_of_lenses,P,f,n2,Zv,Zo);
temp1=1;
fprintf(fv,"\nvertex_level_pixel                pixel hits                lens_vertex_position\n\n");
radius=f*((n2-1)/n2);
c_to_chord=sqrt((radius*radius)-((P*0.5)*(P*0.5)));  /*centre of curvature to sag depth plane */
sag=radius-c_to_chord;
for (pq=0;pq<=(num_of_lenses*P);pq=pq+P)          /* incrementing by lenslet pitch */
  {
    num=0;  Yo=Yo+P;
    Yv=pq;
    for (t=0;t<(arc*2);t=t+1)
       pix[t]=0;
   /* line length from ob. point to centre of curvature – see p.63  */
    pc=sqrt(((Yv-Yo)*(Yv-Yo))+((Zo-Zv-radius)*(Zo-Zv-radius)));

    arc_angle=atan((P*0.5)/c_to_chord);

    for (nnn=0;nnn<=(2*arc);nnn=nnn+1)
       {
         z_chord=Zv+radius-(radius*cos((arc_angle*(arc-nnn))/arc));   /*coords on lens curvature*/
         y_chord=Yv+(radius*sin((arc_angle*(arc-nnn))/arc));
         pt=sqrt(((z_chord-Zo)*(z_chord-Zo))+((Yo-y_chord)*(Yo-y_chord)));
         sigma=acos(((pc*pc)+(pt*pt)-(radius*radius))/(2*pc*pt));        /* cosine rule */

         thet1=asin((pc*sin(sigma))/radius);                          /* Snell's Law */
         thet2=asin(sin(thet1)/n2);
     /* different eqns required for different ob. point orientations */
```

```
            if (y_chord>Yv)  chord_angle=acos((y_chord-Yv)/radius);
           else chord_angle=acos((Yv-y_chord)/radius);
           yrm=(Zv+radius-Zo)/tan(chord_angle);
           if (Yo>y_chord)
            {
              if (y_chord>Yv)    thet3=1.570796-chord_angle-thet2;
               else  thet3=chord_angle+thet2-1.570796;
            }
          if (Yo<y_chord)
            {
              if (y_chord>Yv)    thet3=chord_angle+thet2-1.570796;
              else  thet3=1.570796-chord_angle-thet2;
            }

         if (Yo>(Yv+yrm))
             {
               if (y_chord>Yv)  thet3=1.570796-chord_angle+thet2;
               else   thet3=chord_angle+thet2-1.570796;
             }
         if (Yo<(Yv-yrm))
             {
               if (y_chord<Yv)  thet3=1.570796-chord_angle+thet2;
               else   thet3=chord_angle+thet2-1.570796;
             }

         yr=(Zv+f-z_chord)*tan(thet3);
          if (yr<0) yr=-yr;
          if (y_chord>Yv)
              {
                  if ((chord_angle+thet2)<1.570796)  y_fin=y_chord-yr;
                  else   y_fin=y_chord+yr;
              }
           if (y_chord<Yv)
              {
                   if ((chord_angle+thet2)>1.570796)  y_fin=y_chord-yr;
                   else  y_fin=y_chord+yr;
              }
           if (y_chord==Yv)
              {
                    if (Yo>Yv)  y_fin=y_chord-yr;
                    if (Yo<Yv)  y_fin=y_chord+yr;
                    if (Yo==Yv)    { y_fin=y_chord;   yrm=0;       }
              }

           if (Yo>(Yv+yrm))  y_fin=y_chord-yr;
           if (Yo<(Yv-yrm))  y_fin=y_chord+yr;


            yf=floor((y_fin*sc)+0.5);
           pix[num]=yf;     /* num initialized to 1  */
           num++;
       }
     equiv_Yv=floor((Yv*sc)+0.5);
     for (p=0;p<(arc*2)+1;p=p+1)
        {
         fprintf(fv," %d",pix[p]-equiv_Yv);
         fprintf(fv,"\t %.2f\n",Yv); i++;printf("i=%d",i);
  }
fclose (fv);
return(0);  }
```

Appendix G, Part 1 (see 7.2)

Parses through a VRML2 file and transforms it to integral gem file format

Cyber-sculpture

Parses through a VRML2 file and transforms it to integral GEM file format

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>

#define FOCAL_LENGTH   200.0
#define FIELDLENS_DIST  3000.0
#define NUMBER_OF_TRIANGLES  1440
#define OBJECTS  2


typedef struct {
     float x1;
     float y1;
     float z1;
          } segments;

 typedef struct {
      float x;
      float y;
      float z;
          } vector;

 int stop_coord[OBJECTS];
 float trans[10][3],rots[10][4],diffuseColor[10][3];
 vector triangles[OBJECTS][8*NUMBER_OF_TRIANGLES];

 void write_gem_file();

 int main()
  {
   float x1,y1,z1;
   int tran=0,rot=0,dC=0,which=0,v1,v2,v3;
   int run,nq,minus;
   char parse_str[45];
   segments seg[OBJECTS][NUMBER_OF_TRIANGLES]; /* segment identification */
   FILE *fg;
   fg=fopen("DE010.wrl","r");              /* open vrml2 file */
   if (fg==NULL)
     {
       printf("\n\n\t\t\t****Input File Problem ******** !!!\n\n");  fclose(fg); exit(1);
     }
   printf("\nReading VRML file\n");
   fscanf(fg,"%s",parse_str);
   if (strcmp(parse_str,"#VRML")==0)
     printf("\n#VRML");
    else printf("\n Unrecognised file format");

   fscanf(fg,"%s",parse_str);
   if (strcmp(parse_str,"V2.0")==0)
          printf(" V2.0");
   else printf("\n Unrecognised file format");
   fscanf(fg,"%s",parse_str);
   if (strcmp(parse_str,"utf8")==0)
```

```
        printf(" utf8\n");
      else printf("\n Unrecognised file format");

 while(!feof(fg))                          /* while end of file not reached do.. */
  {
  fscanf(fg,"%s",parse_str);
  if(strcmp(parse_str,"Transform")==0)       /* Find Transform */
  {
   do
     {
      fscanf(fg,"%s",parse_str);
      if(strcmp(parse_str,"translation")==0)
          fscanf(fg,"%f %f %f",&trans[tran][0],&trans[tran][1],&trans[tran++][2]);
      if(strcmp(parse_str,"rotation")==0)
          fscanf(fg,"%f %f %f %f",&rots[rot][0],&rots[rot][1],&rots[rot][2],&rots[rot++][3]);
     }
      while (strcmp(parse_str,"children")!=0);
      if(strcmp(parse_str,"diffuseColor")==0)
        fscanf(fg,"%f %f%f",&diffuseColor[dC][0],&diffuseColor[dC][1],&diffuseColor[dC++][2]);

      if(strcmp(parse_str,"IndexedFaceSet")==0)
        {
         do
           {
              fscanf(fg,"%s",parse_str);
           }
            while (strcmp(parse_str,"point")!=0);
      fscanf(fg,"%s",parse_str);
      run = 0;
      while (fscanf(fg,"%f %f %f",&x1,&y1,&z1)==3)       /* store coords in seg[][] */
         {
         /* u = FIELDLENS_DIST-z1;
           z1 = (FOCAL_LENGTH*u)/(u-FOCAL_LENGTH);  Cancel out these two lines for
           normal operation */
           seg[which][run].x1=x1;
           seg[which][run].y1=y1;
           seg[which][run].z1=z1;
           run++;
         }
          stop_coord[which]=run-1;
          which++;
       }
if (strcmp(parse_str,"coordIndex")==0)    /* Find coordIndex */
   {
    fscanf(fg,"%s",parse_str);
    if(strcmp(parse_str,"[")==0)
       {
         nq=0;
         while(fscanf(fg,"%d %d %d %d",&v1,&v2,&v3,&minus)==4)
              {                                          /* sort coords to make triangles */

                triangles[which-1][nq].x=seg[which-1][v1].x1;
                triangles[which-1][nq].y=seg[which-1][v1].y1;
                triangles[which-1][nq].z=seg[which-1][v1].z1;
                nq++;
                triangles[which-1][nq].x=seg[which-1][v2].x1;
                triangles[which-1][nq].y=seg[which-1][v2].y1;
                triangles[which-1][nq].z=seg[which-1][v2].z1;
                nq++;
                triangles[which-1][nq].x=seg[which-1][v3].x1;
```

```
                    triangles[which-1][nq].y=seg[which-1][v3].y1;
                    triangles[which-1][nq].z=seg[which-1][v3].z1;
                    nq++;
                  }
              }
          }
      }
  fclose(fg);
  write_gem_file();
 }
 void write_gem_file()
   {
    FILE *fp;
    int count,gt;
    fp=fopen("de010.gem","w");

    fprintf(fp,"%s     %d\n","ANIMATION:",1);
    fprintf(fp,"%s      %s\n","SHADING:","GOURAUD");
    fprintf(fp,"%s       %s\n","ARRAY:","LENTICULAR");
    fprintf(fp,"%s    %s\n","PROJECTION:","PERSPECTIVE");
    fprintf(fp,"%s    %f %f %f %f %f\n","WEIGHTINGS:",0.8,0.5,0.6,40.0,0.48);
    fprintf(fp,"%s    %d %d %d\n","BACKGROUND:",0,100,120);
    fprintf(fp,"%s      %f %f %f\n","SOURCE:",0.5,0.5,0.5);
    fprintf(fp,"%s      %f %f %f %f\n","LENSES:",2.116667,1.56,3.43718,1.0);
    fprintf(fp,"%s     %f\n","APERTURE:",337.92);
    fprintf(fp,"%s %d\n","PROJ_LOCATIONS:",16);
    fprintf(fp,"%s    %f\n","ARRAY_DEPTH:",0.5);
    fprintf(fp,"%s      %d %d %d\n","SHIFTS:",1200,1000,201);
    fprintf(fp,"%s    %f\n","OUTPUT_RES:",192.424242);
    fprintf(fp,"%s   %f\n","LINE_DENSITY:",1.0);
    fprintf(fp,"%s   %d %f %f %f\n","INITIAL_SCALE:",1,1.4,1.4,1.4);
    fprintf(fp,"%s      %d %f %d %d %d\n","ROTATION:",0,0.0,0,0,0);
    fprintf(fp,"%s    %d %f %f %f\n","TRANSLATION:",0,0.0, 0.0,0.0);
    fprintf(fp,"%s      %d %f %f %f\n","SCALING:",0,0.0,0.0,0.0);
    fprintf(fp,"%s\n\n","SCENE");

    for (gt=1;gt<=OBJECTS;gt++)
        {
         fprintf(fp,"%s  %d\n","OBJECT",gt);
         fprintf(fp,"%s     %d %f %d %d %d\n","ROTATE_OB:",0,0.0,0,0,0);
         fprintf(fp,"%s   %d %f %f %f\n","TRANSLATE_OB:",0,0.0,0.0,0.0);
         fprintf(fp,"%s      %d %f %f %f\n","SCALE_OB:",0,0.0,0.0,0.0);
         fprintf(fp,"%s  %d\n","COLOUR",gt);
         fprintf(fp,"%s       %f %f %f %f\n","ROTATE:",rots[gt-1][3],rots[gt-1][0],rots[gt-
1][1],rots[gt-1][2]);
         fprintf(fp,"%s      %d %f %f %f\n","TRANSLATE:",1,trans[gt-1][0],trans[gt-1][1],trans[gt-
1][2]);
         fprintf(fp,"%s         %d %f %f %f\n","SCALE:",0,0.0,0.0,0.0);
         fprintf(fp,"%s\n","TRIANGLES:");
         for (count=0;count<stop_coord[gt-1];count+=3)
           {                                          /* write triangle cords */
         fprintf(fp,"%f %f %f %f %f %f %f %f %f\n",triangles[gt-1][count].x,triangles[gt-
1][count].y,triangles[gt-1][count].z,triangles[gt-1][count+1].x,triangles[gt-1][count+1].y,triangles[gt-
1][count+1].z,triangles[gt-1][count+2].x,triangles[gt-1][count+2].y,triangles[gt-1][count+2].z);

        }}
 fprintf(fp,"%s","SCENE END");
 fclose(fp);
}
```

Appendix G, Part 2 (see 7.2)

Integral scene file (gem) set up to produce a 400 frame animation

Integral scene file set up to produce a 400 frame animation in which 2 pairs of torii are scaling up and down, and rotating, in pairs, in opposite directions. Only a very small fraction of the triangle coordinates are presented.

```
ANIMATION:          400
SHADING:            GOURAUD
ARRAY:              LENTICULAR
PROJECTION:         PERSPECTIVE
WEIGHTINGS:         0.8 0.5 0.6 40.0 0.48
BACKGROUND:         0 0 0
SOURCE:             0.5 0.5 0.5
LENSES:             2.116667 1.56 3.43718 1.0
APERTURE:           480
PROJ_LOCATIONS:     51
ARRAY_DEPTH:        0.26
SHIFTS:             1202 1782 201
OUTPUT_RES:         204.016064
LINE_DENSITY:       1.0
INITIAL_SCALE:      1 1.5 1.5 1.5
ROTATION:           0 0.0 0 0 0
TRANSLATION:        0 0.0 0.0 0.0
SCALING:            0 0.0 0.0 0.0
SCENE

OBJECT   1
ROTATE_OB:          0 0.0 0 0 0
TRANSLATE_OB:       0 0.0 0.0 0.0
SCALE_OB:           1 -0.001253 -0.001253 -0.001253
COLOUR              1 88 88 250
ROTATE:             1 -0.45 1 0 0
TRANSLATE:          0 0.0 0.0 0.0
SCALE:              0 1.0 1.0 1.0
TRIANGLES:
-24.527994 66.738007 -48.468002 -26.196795 69.160004 -50.358002 -
25.500994 69.552002 -51.240005
-24.527994 66.738007 -48.468002 -25.500994 69.552002 -51.240005 -
23.837793 67.102005 -49.307999
-23.837793 67.102005 -49.307999 -25.500994 69.552002 -51.240005 -
25.115995 69.692001 -52.136002
-23.837793 67.102005 -49.307999 -25.115995 69.692001 -52.136002 -
23.441595 67.228012 -50.162003
-23.441595 67.228012 -50.162003 -25.115995 69.692001 -52.136002 -
25.096395 69.566010 -52.975998
-23.441595 67.228012 -50.162003 -25.096395 69.566010 -52.975998 -
23.389793 67.102005 -50.973999
-23.389793 67.102005 -50.973999 -25.096395 69.566010 -52.975998 -
25.454794 69.188004 -53.675995
-23.389793 67.102005 -50.973999 -25.454794 69.188004 -53.675995 -
23.701994 66.738007 -51.645996
-23.701994 66.738007 -51.645996 -25.454794 69.188004 -53.675995 -
26.163195 68.600006 -54.151993
-23.701994 66.738007 -51.645996 -26.163195 68.600006 -54.151993 -
24.351593 66.150009 -52.121994
-24.351593 66.150009 -52.121994 -26.163195 68.600006 -54.151993 -
27.145994 67.844009 -54.334000
-24.351593 66.150009 -52.121994 -27.145994 67.844009 -54.334000 -
25.275595 65.422005 -52.332001
-25.275595 65.422005 -52.332001 -27.145994 67.844009 -54.334000 -
28.307993 67.004005 -54.207993
```

```
-25.275595 65.422005 -52.332001 -28.307993 67.004005 -54.207993 -
26.375996 64.596008 -52.248001
-26.375996 64.596008 -52.248001 -28.307993 67.004005 -54.207993 -
29.511995 66.150009 -53.759995
-26.375996 64.596008 -52.248001 -29.511995 66.150009 -53.759995 -
27.523996 63.784008 -51.869995
-27.523996 63.784008 -51.869995 -29.511995 66.150009 -53.759995 -
30.631994 65.380005 -53.059998
-27.523996 63.784008 -51.869995 -30.631994 65.380005 -53.059998 -
28.601995 63.042004 -51.225998
-28.601995 63.042004 -51.225998 -30.631994 65.380005 -53.059998 -
31.541996 64.764008 -52.177994
-28.601995 63.042004 -51.225998 -31.541996 64.764008 -52.177994 -
29.497995 62.440010 -50.414001
-29.497995 62.440010 -50.414001 -31.541996 64.764008 -52.177994 -
32.157997 64.358009 -51.197998
-29.497995 62.440010 -50.414001 -32.157997 64.358009 -51.197998 -
30.113995 62.062008 -49.490005
-30.113995 62.062008 -49.490005 -32.157997 64.358009 -51.197998 -
32.451996 64.218002 -50.246002
-30.113995 62.062008 -49.490005 -32.451996 64.218002 -50.246002 -
30.421995 61.936008 -48.593994
-30.421995 61.936008 -48.593994 -32.451996 64.218002 -50.246002 -
32.381996 64.344009 -49.419998
-30.421995 61.936008 -48.593994 -32.381996 64.344009 -49.419998 -
30.393995 62.062008 -47.795998
-30.393995 62.062008 -47.795998 -32.381996 64.344009 -49.419998 -
31.975996 64.722000 -48.804001
-30.393995 62.062008 -47.795998 -31.975996 64.722000 -48.804001 -
30.029997 62.440010 -47.166000
-30.029997 62.440010 -47.166000 -31.975996 64.722000 -48.804001 -
31.289993 65.338013 -48.425995
-30.029997 62.440010 -47.166000 -31.289993 65.338013 -48.425995 -
29.399996 63.042004 -46.787994
-29.399996 63.042004 -46.787994 -31.289993 65.338013 -48.425995 -
30.365993 66.094009 -48.328003
-29.399996 63.042004 -46.787994 -30.365993 66.094009 -48.328003 -
28.545996 63.784008 -46.662003
-28.545996 63.784008 -46.662003 -30.365993 66.094009 -48.328003 -
29.315994 66.948006 -48.509995
-28.545996 63.784008 -46.662003 -29.315994 66.948006 -48.509995 -
27.537994 64.596008 -46.787994
-27.537994 64.596008 -46.787994 -29.315994 66.948006 -48.509995 -
28.195993 67.788010 -48.930008
-27.537994 64.596008 -46.787994 -28.195993 67.788010 -48.930008 -
26.473993 65.422005 -47.166000
-26.473993 65.422005 -47.166000 -28.195993 67.788010 -48.930008 -
27.131994 68.558006 -49.559998
-26.473993 65.422005 -47.166000 -27.131994 68.558006 -49.559998 -
25.435194 66.150009 -47.739998
-25.435194 66.150009 -47.739998 -27.131994 68.558006 -49.559998 -
26.196795 69.160004 -50.358002
-25.435194 66.150009 -47.739998 -26.196795 69.160004 -50.358002 -
24.527994 66.738007 -48.468002
-26.196795 69.160004 -50.358002 -27.831993 71.666000 -52.037994 -
27.103994 72.100006 -53.017990
-26.196795 69.160004 -50.358002 -27.103994 72.100006 -53.017990 -
25.500994 69.552002 -51.240005
-25.500994 69.552002 -51.240005 -27.103994 72.100006 -53.017990 -
26.711994 72.282013 -54.026001
```

```
-25.500994 69.552002 -51.240005 -26.711994 72.282013 -54.026001 -
25.115995 69.692001 -52.136002
-25.115995 69.692001 -52.136002 -26.711994 72.282013 -54.026001 -
26.725992 72.156006 -54.936005
-25.115995 69.692001 -52.136002 -26.725992 72.156006 -54.936005 -
25.096395 69.566010 -52.975998
```

Appendix H (see 7.6)

Parser/translater to change VRML2 format, with H-anim Protos, to integral scene file format, including texture mapping

Parser/translater to change VRML2 format, with H-anim Protos, to integral scene file format – to include texture mapping

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>
#include <ppm.h>

#define size 20000
#define TEXWIDTH 1200     /* width and height of texture map */
#define TEXHEIGHT 2100
#define PIE 3.141593

 typedef struct {
          float x;
          float y;
          float z;
          } vector;

 typedef struct {
          float s;
          float t;
          } texels;          /* texmap cords */

typedef struct {
     float red;
     float green;
     float blue;
          } colour;

typedef struct {
     float x1;
     float y1;
     float z1;
          } segments;

int fileppm_red[TEXWIDTH][TEXHEIGHT];
int fileppm_green[TEXWIDTH][TEXHEIGHT];
int fileppm_blue[TEXWIDTH][TEXHEIGHT];

void write_new_object_file(FILE *fp,char *url_str,int twidth,int theight);
void command_error();
void add_ext(char *fname,char *front,char *ext,int force);

int main(int argc,char *argv[])
 {
  int n,nn,t,texsize=0,v1,v2,v3,r,g,b,r1,g1,b1,r2,g2,b2,r3,g3,b3,aa=0,w,h,gee;
  int horiz1,vert1,horiz2,vert2,horiz3,vert3,ff,re,gg,number;
  int which=0,search,hold,run,all_i,stop_coord[30],nq,length,i,twidth,theight;
  float x1,y1,z1,u,v,x_temp,deg1,deg2,thet1,thet2;
  float temp_u,temp_v;
  vector triangles[size];
  texels tex_point[size],tex_triangles[size];
  char col_str[45],seg_str[45],comma[1],com,seg_name[30][40],url_str[45];
  char input_file[50]="", output_file[50]="";
  segments seg[30][1000],all_segs[20000];
```

```
 FILE *fg,*fp,*fd;
 if (argc<2) command_error();
 strcpy (input_file, argv[1]);

 if (argc==2)
  {
  strcpy (output_file, input_file);
  add_ext(output_file,"gem_input/", "gem",1);
  }
 else
  {
  strcpy (output_file, argv[2]);
  add_ext(output_file,"gem_input/", "gem",1);
  printf("\nargc = %d\n",argc);
  }
 add_ext(input_file,"prometheus_textures/","wrl",0);

 fg=fopen(input_file,"r");
 if (fg==NULL)
  {
  printf("\n\n\t\t\t****Input File Problem ******** !!!\n\n");  fclose(fg); exit(1);
  }
 printf("\nReading VRML file\n");
 fscanf(fg,"%s",col_str);
 if (strcmp(col_str,"#VRML")==0)
     printf("\n#VRML");
 else printf("\n Unrecognised file format");

 fscanf(fg,"%s",col_str);
 if (strcmp(col_str,"V2.0")==0)
   printf(" V2.0");
 else printf("\n Unrecognised file format");

 fscanf(fg,"%s",col_str);
 if (strcmp(col_str,"utf8")==0)
   printf(" utf8\n");
 else printf("\n Unrecognised file format");

 printf("\n  argc = %d\n",argc); printf("\n");

 deg1=180.0; thet1=(3.141593/180.0)*deg1;
 deg2=90.0; thet2=(3.141593/180.0)*deg2;

while(!feof(fg))
{
 fscanf(fg,"%s",col_str);
 if(strcmp(col_str,"DEF")==0)                          /* Find DEF and Segment */
  {
  fscanf(fg,"%s",seg_str);
  fscanf(fg,"%s",col_str);
  if(strcmp(col_str,"Segment")==0)
   {                                    /* store segment name in seg_name[] */
    strcpy(seg_name[which],seg_str); /* printf("\n %s",seg_name[which]);  */
   do
    {                                   /* search for word 'point' */
    fscanf(fg,"%s",col_str);
    }
   while (strcmp(col_str,"point")!=0);
   fscanf(fg,"%s",col_str);
   run = 0;
```

```
      while (fscanf(fg,"%f %f %f",&x1,&y1,&z1)==3)
       {
          x_temp=x1;
          x1=(x1*cos(thet1))+(z1*sin(thet1));
          z1=-(x_temp*sin(thet1))+(z1*cos(thet1));
          x_temp=x1;
         x1=(x1*cos(thet2))-(y1*sin(thet2));
         y1=(x_temp*sin(thet2))+(y1*cos(thet2));

       seg[which][run].x1=x1;          /* store cords */
       seg[which][run].y1=y1;
       seg[which][run].z1=z1;
       run++;
      }
     stop_coord[which]=run-1;
     which++;
    }
  }


if (strcmp(col_str,"segments")==0)                    /* Find segments - hence USE name */
 {
  fscanf(fg,"%s",col_str);
  hold=0;
  do
   {
    fscanf(fg,"%s",col_str);
    fscanf(fg,"%s",seg_str);
    for(search=0;search<which;search++)
     {
      if (strcmp(seg_str,seg_name[search])==0)
       {
        for(all_i=0;all_i<=stop_coord[search];all_i++)
          {
           all_segs[hold+all_i].x1=seg[search][all_i].x1;
           all_segs[hold+all_i].y1=seg[search][all_i].y1;
           all_segs[hold+all_i].z1=seg[search][all_i].z1;
          }
       hold=hold+all_i;
       }
     }
   }
  while(strcmp(col_str,"]")!=0);
 }
if (strcmp(col_str,"texture")==0)                     /* Find texture hence url - then texture map */
  {
  fscanf(fg,"%s",col_str);
  fscanf(fg,"%s",col_str);
  fscanf(fg,"%s",col_str);
  if (strcmp(col_str,"url")==0)
   {
    fscanf(fg,"%s",url_str);
    length=strlen(url_str);

    for (i=0; i<length;i++)
     {
      if (url_str[i]=='.') break;
      url_str[i]=url_str[i+1];
     }
    strcpy (&url_str[i],"ppm");
```

```
}
        }
  if (strcmp(col_str,"coordIndex")==0)                    /* Find coordIndex  */
     {
      fscanf(fg,"%s",col_str);
      if(strcmp(col_str,"[")==0)
       {
        nq=0;
        while(fscanf(fg,"%d %d %d %s",&v1,&v2,&v3,comma)==4)
         {        /*store triangles in order of 'USE'*/
        triangles[nq].x=all_segs[v1].x1; triangles[nq].y=all_segs[v1].y1;
        triangles[nq++].z=all_segs[v1].z1;
        triangles[nq].x=all_segs[v2].x1; triangles[nq].y=all_segs[v2].y1;
        triangles[nq++].z=all_segs[v2].z1;
        triangles[nq].x=all_segs[v3].x1; triangles[nq].y=all_segs[v3].y1;
        triangles[nq++].z=all_segs[v3].z1;
         }
       }
     }


   if (strcmp(col_str,"texCoord")==0)                     /* Find texCoord hence point */
    {
     fscanf(fg,"%s",col_str);
     fscanf(fg,"%s",col_str);
     fscanf(fg,"%s",col_str);
     if (strcmp(col_str,"point")==0)
      {
       fscanf(fg,"%s",col_str);
       temp_u=0.0; temp_v=0.0;
       while (fscanf(fg,"%f %f %c",&u,&v,&com)==3)      /* run through to find max u and v */
        {
         if (u>temp_u) temp_u=u; if (v>temp_v) temp_v=v;
         tex_point[texsize].s=u;
       tex_point[texsize++].t=v;
        }
      }
    }

   if (strcmp(col_str,"texCoordIndex")==0)                /* Find texCoordIndex hence [ */
    {
     fscanf(fg,"%s",col_str);
     if (strcmp(col_str,"[")==0)
      {
       nn=0;    /*read in and store coord index for texmap*/
       while (fscanf(fg,"%d %d %d %s",&v1,&v2,&v3,comma)==4)
        {
         tex_triangles[nn].s=tex_point[v1].s; tex_triangles[nn++].t=tex_point[v1].t;
         tex_triangles[nn].s=tex_point[v2].s; tex_triangles[nn++].t=tex_point[v2].t;
         tex_triangles[nn].s=tex_point[v3].s; tex_triangles[nn++].t=tex_point[v3].t;
        }
      }
    }
  }


 fclose (fg);

/*****************************************************************************/
```

```
fp=fopen(output_file,"w");
 if (fp==NULL)
       {
        printf("\n\n\t\t\tWriting to ava???.gem File Problem   !!!\n\n");  fclose(fp); exit(1);
       }
 fd=fopen(url_str,"r"); printf("\nurl string = %s",url_str);
/*fd=fopen("prometheus_textures/avatar342.ppm","r");*/
  if (fd==NULL)
      {
       printf("\n\n\t\t\tavatar texmap File Problem   !!!\n\n");  fclose(fd); exit(1);
      }

     fscanf(fd,"%s",col_str);                    /* check texmap is in correct format */
      if (strcmp(col_str,"P3")==0)
         printf("\nP3");

     fscanf(fd,"%s",col_str);  printf("\n%s",col_str);
     fscanf(fd,"%s",col_str); printf(" %s",col_str);
     fscanf(fd,"%s",col_str); printf(" %s",col_str);
     fscanf(fd,"%s",col_str); printf(" %s",col_str);
     fscanf(fd,"%s",col_str); printf(" %s",col_str);
     fscanf(fd,"%s",col_str); printf(" %s",col_str);
     fscanf(fd,"%s",col_str); printf(" %s",col_str);
     fscanf(fd,"%d",&twidth); printf("\n%d",twidth);
     fscanf(fd,"%d",&theight); printf(" %d\n",theight);
     fscanf(fd,"%s",col_str); printf("%s\n\n",col_str);
     w=0; h=theight-1;
     while (fscanf(fd,"%d %d %d",&r,&g,&b)==3)  /* read in texmap cords */
      {
          fileppm_red[w][h]=r;
          fileppm_green[w][h]=g;
          fileppm_blue[w++][h]=b;
          if (w==twidth+1) { w=0; h--; }
      }


    write_new_object_file(fp,url_str,twidth,theight);

    gee=0;
    for (re=0;re<nq;re+=3)
     {
      gg=0;
      horiz1=(int)floor(tex_triangles[re].s*(twidth/temp_u));   /*derive texmap */
      vert1=(int)floor(tex_triangles[re].t*(theight/temp_v));
      horiz2=(int)floor(tex_triangles[re+1].s*(twidth/temp_u));
      vert2=(int)floor(tex_triangles[re+1].t*(theight/temp_v));
      horiz3=(int)floor(tex_triangles[re+2].s*(twidth/temp_u));
      vert3=(int)floor(tex_triangles[re+2].t*(theight/temp_v));

         r1=fileppm_red[horiz1][vert1];
         g1=fileppm_green[horiz1][vert1];
         b1=fileppm_blue[horiz1][vert1];
         r2=fileppm_red[horiz2][vert2];
         g2=fileppm_green[horiz2][vert2];
         b2=fileppm_blue[horiz2][vert2];
         r3=fileppm_red[horiz3][vert3];
         g3=fileppm_green[horiz3][vert3];
         b3=fileppm_blue[horiz3][vert3];

          aa++;
```

```
    fprintf(fp,"%s %d   %d %d %d %d %d %d %d %d %d\n","COLOUR
",aa,r1,g1,b1,r2,g2,b2,r3,g3,b3);
    fprintf(fp,"%s    %d %d %d %d %d %d\n","TEX_MAP ",horiz1,vert1,horiz2,vert2,horiz3,vert3);
    fprintf(fp,"%s        %d %d %d %d %d\n","ROTATE: ",0,20,0,0,0);
    fprintf(fp,"%s      %d %f %f %f\n","TRANSLATE: ",0,0.0,0.0,0.0);
    fprintf(fp,"%s        %d %f %f %f\n","SCALE: ",0,1.0,1.0,1.0);
    fprintf(fp,"%s\n","TRIANGLES: ");
    fprintf(fp,"%f %f %f %f %f %f %f %f %f \n",
    triangles[gee].x,triangles[gee].y,triangles[gee].z,
    triangles[gee+1].x,triangles[gee+1].y,triangles[gee+1].z,
    triangles[gee+2].x,triangles[gee+2].y,triangles[gee+2].z);
    gee=gee+3;
     }

fprintf(fp,"\n%s %s","SCENE","END");
fclose(fp); fclose(fd);
return(0);
}

void write_new_object_file(FILE *fp, char *url_str,int twidth,int theight)
  {
   FILE *fz;
   int animate,bkgr,bkgg,bkgb,Iir,Iig,Iib,aperture,trirays,vert_shift,horiz_shift,z_shift;
   int con_rot1,Rx1,Ry1,Rz1,con_tr1,con_sc1,con_sc0;
   float
Ka,Kd,Ks,N,brightness,pitch,n2,f,n1,zv_pos,dpi,space_control,Sx0,Sy0,Sz0,theta1,Tx1,Ty1,Tz1,Sx1,S
y1,Sz1;
   char in_str[30],shading[30],lens_array[30],projection[30];

   fz=fopen("gem_input/param_list.lst","r");

 while(!feof(fz))
  {
    fscanf(fz,"%s",in_str);
    if (strcmp(in_str,"ANIMATION:")==0)
     fscanf(fz,"%d",&animate);
    if (strcmp(in_str,"SHADING:")==0)
     fscanf(fz,"%s",shading);  /* this should be FLAT, GOUROUD  or PHONG  */
    if (strcmp(in_str,"ARRAY:")==0)
     fscanf(fz,"%s",lens_array);   /* this should be LENTICULAR or MICROLENS  */
    if (strcmp(in_str,"PROJECTION:")==0)
     fscanf(fz,"%s",projection);  /* this should be ORTHOGONAL or PERSPECTIVE  */
    if (strcmp(in_str,"WEIGHTINGS:")==0)
      fscanf(fz,"%f %f %f %f %f",&Ka,&Kd,&Ks,&N,&brightness);
    if (strcmp(in_str,"BACKGROUND:")==0)
      fscanf(fz,"%d %d %d",&bkgr,&bkgg,&bkgb);
    if (strcmp(in_str,"SOURCE:")==0)
      fscanf(fz,"%d %d %d",&Iir,&Iig,&Iib);
    if (strcmp(in_str,"LENSES:")==0)
      fscanf(fz,"%f %f %f %f",&pitch,&n2,&f,&n1);
    if (strcmp(in_str,"APERTURE:")==0)
       fscanf(fz,"%d",&aperture);
    if (strcmp(in_str,"PROJ_LOCATIONS:")==0)
       fscanf(fz,"%d",&trirays);
    if (strcmp(in_str,"ARRAY_DEPTH:")==0)
       fscanf(fz,"%f",&zv_pos);
    if (strcmp(in_str,"SHIFTS:")==0)
       fscanf(fz,"%d %d %d",&vert_shift,&horiz_shift,&z_shift);
    if (strcmp(in_str,"OUTPUT_RES:")==0)
      fscanf(fz,"%f",&dpi);
```

```
        if (strcmp(in_str,"LINE_DENSITY:")==0)
          fscanf(fz,"%f",&space_control);
        if (strcmp(in_str,"INITIAL_SCALE:")==0)
          fscanf(fz,"%d %f %f %f",&con_sc0,&Sx0,&Sy0,&Sz0);
        if (strcmp(in_str,"ROTATION:")==0)
          { fscanf(fz,"%d %f %d %d %d",&con_rot1,&theta1,&Rx1,&Ry1,&Rz1); /*
   theta1=theta1*PIE/180;*/ }
        if (strcmp(in_str,"TRANSLATION:")==0)
           fscanf(fz,"%d %f %f %f",&con_tr1,&Tx1,&Ty1,&Tz1);
        if (strcmp(in_str,"SCALING:")==0)
          fscanf(fz,"%d %f %f %f",&con_sc1,&Sx1,&Sy1,&Sz1);
      }
        fclose(fz);

      fprintf(fp,"%s       %d\n","ANIMATION:",animate);
      fprintf(fp,"%s        %s\n","SHADING:",shading);
      fprintf(fp,"%s         %s\n","ARRAY:",lens_array);
      fprintf(fp,"%s      %s\n","PROJECTION:",projection);
      fprintf(fp,"%s      %f %f %f %f %f\n","WEIGHTINGS:",Ka,Kd,Ks,N,brightness);
      fprintf(fp,"%s      %d %d %d\n","BACKGROUND:",bkgr,bkgg,bkgb);
      fprintf(fp,"%s        %d %d %d\n","SOURCE:",Iir,Iig,Iib);
      fprintf(fp,"%s        %f %f %f %f\n","LENSES:",pitch,n2,f,n1);
      fprintf(fp,"%s       %d\n","APERTURE:",aperture);
      fprintf(fp,"%s    %d\n","PROJ_LOCATIONS:",trirays);
      fprintf(fp,"%s     %f\n","ARRAY_DEPTH:",zv_pos);
      fprintf(fp,"%s        %d %d %d\n","SHIFTS:",vert_shift,horiz_shift,z_shift);
      fprintf(fp,"%s      %f\n","OUTPUT_RES:",dpi);
      fprintf(fp,"%s     %f\n","LINE_DENSITY:",space_control);
      fprintf(fp,"%s    %d %f %f %f\n","INITIAL_SCALE:",con_sc0,Sx0,Sy0,Sz0);
      fprintf(fp,"%s       %d %f %d %d %d\n","ROTATION:",con_rot1,theta1,Rx1,Ry1,Rz1);
      fprintf(fp,"%s     %d %f %f %f\n","TRANSLATION:",con_tr1,Tx1,Ty1,Tz1);
      fprintf(fp,"%s       %d %f %f %f\n\n","SCALING:",con_sc1,Sx1,Sy1,Sz1);
      fprintf(fp,"%s\n","SCENE");
      fprintf(fp,"%s   %s\n","TEX_NAME ",url_str);
      fprintf(fp,"%s   %d %d\n","TEX_SIZE ",twidth,theight);
    }

  void command_error()
    {
      printf("\nUsage:  infile[.wrl]  [outfile[.gem]]\n\n");
      exit(1);
    }

  void add_ext(char *fname,char *front,char *ext,int force)
    {
      int i,length;


      length=strlen(fname);

      for (i=0; i<length;i++)
        if (fname[i]=='.') break;

      if ((fname[i]=='\0')||(force==1))  /* for output file */
          {
           if (strlen(ext)>0)
          /* strcat (front,t);*/
           fname[i++]='.';
          strcpy (&fname[i],ext);
          strcat (front,fname);
```

```
    strcpy (fname,front);
   }
  else                    /* for input_file  */
   {
    if (strlen(ext)>0)
    fname[i++]='.';
    strcpy (&fname[i],ext);
    strcat (front,fname);
    strcpy (fname,front);
   }
}
```

Appendix I, Part 1 (see 8.2.2)

Production of position and orientation parameters to enter into an object file (Set for IBM LCD QUXGA).

Set for IBM LCD QUXGA and in comments values for Samsung: using 3*pixperlens and 2*pixperlens respectively with 12 lines/in array sheet

This program calculates the three variables for 3Dfrom2D produced the slow way with screen capture.

The variables are:

1) **X-translation i.e. viewpoint positions on the aperture (remember this is on a horizontal line hence LENGTH of display) also remember that there is one more viewpoint than spaces between viewpoints.**
2) **angle subtended by the viewpoint at the central lenslets centre of curvature**
3) **distance of the line from viewpoint to the centre of curvature of the central lenslet.**

TO USE:

a) **precalculate number of viewpoints required i.e. (inter number * pixperlens)**
b) **put in params lenspitch, width and height of display (i.e. number of pixels), pixelpitch and current camera position/orientation values in the VRML2 (MPEGBIFS) file.**

A file is produced (params.txt) which gives the values of position and orientation for different *camera positions. All that is required is to directly insert this* (copy/paste) *into the object file* (e.g. curve2.wrl) and delete the # for each one in turn. For each one in turn run the wrl image produced and screen grab using Print Screen button on keyboard. This is now saved in the clipboard. Paste the image into a viewer (e.g. LView Pro) and save as ppm. Run interlac6b.c suitably set up for that number of viewpoints

CCIP could be a very crucial number.

Note: The distance Zpos is calculated to the focal plane whilst the angle of orientation is calculated at the centre-of-curvature - purely because there is the relevant lengths known to do it this way. The *important thing* is that Zpos is calculated to the capture plane itself

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>


#define LENSPITCH  2.116667
#define WIDTH 3840      /* 1280 */
#define HEIGHT 2400     /* 1024 */
#define PIXELPITCH      0.1245 /* 0.264 */
#define APERTURE WIDTH*PIXELPITCH
#define DPI 25.4/PIXELPITCH
#define CCIP  3.3                               /* centre-of-curvature to image plane */
/* #define ZOB  196.58  object projecting in front of display by this amount - not required for program
 it is decided, pre-program, what amount of the object is required to be replayed in front of display
 to then calculate the number of viewpoints required  */
#define VIEWPOINTS 51           /*16*/
#define currentX 0.0        /*  X, Y and Z positions already defined in object file for position  */
#define currentY 0.0
#define currentZ 0.0
#define currentOrientation 0.0
```

```
void main()
 {
        float D,spacings,Xpos=0,Zpos,orientation;
        int i;
        FILE *fm;

        fm=fopen("params.txt","w");

/*Finding D, distance of APERTURE to LENS ARRAY centre-of-curvature */

        D = (APERTURE*CCIP)/LENSPITCH;

/*Finding Number of viewpoints for II for a scene projecting out from the display is thus:*/

VIEWPOINTS = (APERTURE*ZOB)/(LENSPITCH*(D-ZOB));

/*but this is better for the interlacing  if it is an "integer number * number-of-pixperlens", so is better to
be calculated outside the program and #defined manually. For the case of the TR221 QUXGA IBM
LCD at 17 pixperlens for lens array of 12 lines/in 51 viewpoints allows II up to ZOB i.e.196.58mm
(7.7in) in front of display:

        i.e. 51 = (APERTURE*196.58)/(0.1245*(D-196.58))     */


/* Finding the spacings required between viewpoints  */

spacings = APERTURE/(VIEWPOINTS-1); /*one less space than viewpoints  -VIEWPOINTS-1  */

printf("%s%d %s aperture=%f D=%f\n","#",VIEWPOINTS,"VIEWS",APERTURE,D);
fprintf(fm,"%s%d %s\n","#",VIEWPOINTS,"VIEWS");

 for (i=0;i<=VIEWPOINTS-1;i++)
        {
          if (i==0)         Xpos = Xpos-(APERTURE*0.5)+currentX;
          else Xpos = Xpos+spacings;

/* the fustrum created for each view will set the capture plane square to the viewpoint camera therefore
the camera's Zpos will be its distance to a swivelled centre-of-curvature of the central lenslet */

          Zpos = sqrt((Xpos*Xpos)+(D*D))+CCIP+currentZ;
          orientation = atan(Xpos/Zpos)+currentOrientation;
          printf("%s%d %f %f %f %s %d %d %d %f\n","#DEF
                V",i+1,Xpos,currentY,Zpos,"orientation",0,1,0,orientation);
          fprintf(fm,"%s%d %s %s %s %f %f %f %s %d %d %d %f %s\n","#DEF
V",i+1,"Viewpoint","{","position",Xpos,currentY,Zpos,"orientation",0,1,0,orientation,"}");
        }


        fclose(fm);
```

Appendix I, Part 2 (see 8.2.2)

3D-from-2D multiplexing code - additive intensity is built up in the pixels for anti-aliasing purposes

3D-from-2D multiplexing code - additive intensity is built up in the pixels for anti-aliasing purposes

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>

#define WIDTH 1280
#define HEIGHT 1024
#define PIE 3.141593
#define TRIRAYS 16


int fileppm_red[WIDTH][HEIGHT];
int fileppm_green[WIDTH][HEIGHT];
int fileppm_blue[WIDTH][HEIGHT];
int view;

void scanppm(FILE *fd,FILE *ft);
void write_composite(FILE *ft);

void main()
{
  FILE *fd,*ft;

  ft=fopen("new2.ppm","w");


  fd=fopen("Clipp16.ppm","r"); view=1;
        scanppm(fd,ft);
  fclose(fd);

  fd=fopen("Clipp15.ppm","r"); view=2; printf("\nScanning view 2");
        scanppm(fd,ft);
  fclose(fd);

  fd=fopen("Clipp14.ppm","r"); view=3; printf("\nScanning view 3");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp13.ppm","r"); view=4; printf("\nScanning view 4");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp12.ppm","r"); view=5; printf("\nScanning view 5");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp11.ppm","r"); view=6; printf("\nScanning view 6");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp10.ppm","r"); view=7; printf("\nScanning view 7");
        scanppm(fd,ft);
  fclose(fd);
```

```
        fd=fopen("Clipp9.ppm","r"); view=8; printf("\nScanning view 8");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp8.ppm","r"); view=9; printf("\nScanning view 9");
        scanppm(fd,ft);
  fclose(fd);

  fd=fopen("Clipp7.ppm","r"); view=10; printf("\nScanning view 10");
        scanppm(fd,ft);
  fclose(fd);

  fd=fopen("Clipp6.ppm","r"); view=11; printf("\nScanning view 11");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp5.ppm","r"); view=12; printf("\nScanning view 12");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp4.ppm","r"); view=13; printf("\nScanning view 13");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp3.ppm","r"); view=14; printf("\nScanning view 14");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp2.ppm","r"); view=15; printf("\nScanning view 15");
        scanppm(fd,ft);
  fclose(fd);

        fd=fopen("Clipp1.ppm","r"); view=16; printf("\nScanning view 16\n\n");
        scanppm(fd,ft);
  fclose(fd);


  write_composite(ft);

  fclose(ft);

}


void scanppm(FILE *fd,FILE *ft)
 {
  int w,h,r,g,b,temp_w,twidth,theight,temp_view;
  char col_str[30];


   fscanf(fd,"%s",col_str);
   if (strcmp(col_str,"P3")==0)
     {
       if (view==1) { printf("\nP3");  fprintf(ft,"%s",col_str);
     }
 }

fscanf(fd,"%s",col_str); if (view==1) { printf("\n%s",col_str); fprintf(ft,"\n%s",col_str);}
fscanf(fd,"%s",col_str); if (view==1) { printf(" %s",col_str);  fprintf(ft," %s",col_str); }
fscanf(fd,"%s",col_str); if (view==1) { printf(" %s",col_str);  fprintf(ft," %s",col_str); }
```

```
    fscanf(fd,"%s",col_str); if (view==1) { printf(" %s",col_str);  fprintf(ft," %s",col_str); }
    fscanf(fd,"%s",col_str); if (view==1) { printf(" %s",col_str);  fprintf(ft," %s",col_str); }
    fscanf(fd,"%d",&twidth); if (view==1) { printf("\n%d",twidth);  fprintf(ft,"\n%d",twidth); }
    fscanf(fd,"%d",&theight); if (view==1) { printf(" %d\n",theight); fprintf(ft," %d\n",theight); }
    fscanf(fd,"%s",col_str); if (view==1) { printf("%s\n\n",col_str); fprintf(ft,"%s\n\n",col_str); }
    if (view==1) printf("\nScanning view 1");

    w=-1; h=0;
    while (fscanf(fd,"%d %d %d",&r,&g,&b)==3)
      {
        w++;
        if (w>WIDTH-1) { w=0; h++; }
        temp_w=(WIDTH-1)-w;
        if ((view>0)&&(view<=(TRIRAYS/8.0)))
              temp_view = 0;                                  /* anti-aliasing set up */
  if ((view>(TRIRAYS/8.0))&&(view<=((TRIRAYS*2)/8.0)))              temp_view = 1;
  if ((view>((TRIRAYS*2)/8.0))&&(view<=((TRIRAYS*3)/8.0)))          temp_view = 2;
  if ((view>((TRIRAYS*3)/8.0))&&(view<=((TRIRAYS*4)/8.0)))          temp_view = 3;
  if ((view>((TRIRAYS*4)/8.0))&&(view<=((TRIRAYS*5)/8.0)))          temp_view = 4;
  if ((view>((TRIRAYS*5)/8.0))&&(view<=((TRIRAYS*6)/8.0)))          temp_view = 5;
  if ((view>((TRIRAYS*6)/8.0))&&(view<=((TRIRAYS*7)/8.0)))          temp_view = 6;
  if ((view>((TRIRAYS*7)/8.0))&&(view<=((TRIRAYS*8)/8.0)))          temp_view = 7;

  if (((temp_w-temp_view)%8==0)&&(temp_w<WIDTH)&&(temp_w>=0)) /*filter */
      {
        fileppm_red[temp_w][h]=r;
        fileppm_green[temp_w][h]=g;
        fileppm_blue[temp_w][h]=b;
       }

    }
  }

 void write_composite(FILE *ft)
  {
   int w,h,count=0;

  for (h=0;h<HEIGHT;h++)
   for (w=0;w<WIDTH;w++)
     {
     count++;
     fprintf(ft,"%d %d %d ",fileppm_red[w][h],fileppm_green[w][h],fileppm_blue[w][h]);
     if (count==5) { count=0; fprintf(ft,"\n"); }
     }
}
```

R.F.Stevens, N.Davies, G.Milnthorpe, "Lens arrays and optical system for orthoscopic three-dimensional imaging", Imaging Science Journal 49(3), pp.151-164, 2001

G.Milnthorpe,M.McCormick,N.Davies, "Computer modeling of lens arrays for integral image rendering", Proc. 20th Eurographics UK Conference,136-141,June 2002

J.K.Brown, M.McCormick, N.Davies, M.Forman, G.Milnthorpe, R.Kotecha, "The use of computer generated integral imaging to visualise cyber-structure", Proc. 20th Eurographics UK Conference,3-8, June 2002

M.McCormick, N.Davies, G.Milnthorpe, A.Aggoun, M.Forman, "Integral imaging as a modality for 3D TV and displays", Proc SPIE Vol 4864 59, Boston, 2002

G.Milnthorpe, M.McCormick, A.Aggoun, N.Davies, M.Forman, "Computer generated content for 3D TV displays", IBC 2002, Amsterdam

M.Price, J.Chandaria, O.Grau, G.A.Thomas, D.Chatting, J.Thorne, G.Milnthorpe, P.Woodward, L.Bull, E-J.Ong, A.Hilton, J.Mitchelson, J.Starck, "Real-time production and delivery of 3D media", IBC 2002, Amsterdam

[1] E.H.Adelson, J.R.Bergen, "The plenoptic function and the elements of early vision", *Computational Models of Visual Processing*, Cambridge MA:MIT Press, 1991

[2] T.Okoshi, "Three-Dimensional Imaging Techniques", *Academic Press*, New York, 1976 {a translation and extension of the Japanese "Sanjigen-Gazo Kogaku", *Sangyo-Toshu*, Tokyo, 1972]

[3]

[4]

[5]

[6] T.Yamazaki, K.Kamijo, S.Fukuzumi, "Quantitative evaluation of visual fatigue", *Proc. Japan Display*, pp. 606-609, 1984

[7] R.Borner, "Autostereoscopic photography and video projection of parallax-stereo-panoramagrms onto large lenticular screens", *Japan Display 89*, pp40-43, 1989

[8] A.L.Travis, S.R.Land, "The design and evaluation of a CRT-based autostereoscopic 3-D display", *Proc.SID Vol 32/4*, pp279-283, 1991

[9] N.Tetsutani, K.Omura, F.Kishino, "Wide-screen autostereoscopic display system employing head-position tracking", *Optical Engineering*, 33(11), pp3690-3697, 1994

[10] S.Pastoor, "3D-television: A survey of recent results on subjective requirements", *Signal Processing: Image Communication 4*, pp 21-32, 1991

[11] D.Ezra, B.Omar, G.Woodgate, "Autostereoscopic directional display apparatus", *European Patent Application* EP-A-0 602 934, 1992

[12] D.Ezra, G.J.Woodgate,B.A.Omar, N.S.Holliman, J.Harrold, L.S.Shapiro, "New autostereoscopic display system", *Proc. SPIE Vol 2409*, pp31-40, Feb. 1995

[13] G.J.Woodgate, D.Ezra, J.Harrold, N.S.Holliman, G.R.Jones, R.Moseley, "Observer tracking autostereoscopic 3D display systems", *Proc. SPIE Vol. 3012*, pp187-198, Feb. 1997

[14] J.Harrold, A.Jacobs, G.J.Woodgate, D.Ezra, "Performance of a Convertible, 2D and 3D Parallax Barrier Autosteroscopic Display", *Proc. SID, 20th International Display Research Conference*, Florida, Sept. 2000

[15] N.A.Dodgson, J.R.Moore, S.R.Lang, G.Martin, P.Canepa, "A 50" time-multiplexed autostereoscopic display", *Proc. SPIE 3957, SPIE Symposium on Stereoscopic Displays and Applications XI*, San Jose, California, Jan' 2000

[16] Cees Van Berkel, "Image preparation for 3D-LCD", *Proc. of SPIE Vol. 3639 Stereoscopic ~displays and Virtual Reality Systems VI,* May 1999

[17] Cees Van Berkel, J.A.Clarke, "Characterization and optimisation of 3D-LCD module design", *Proc. SPIE vol 3012* pp179-187, 1997

[18] H.Isono, M.Yasuda, D.Takemori, H.Kanayama, C.Yamada, K.Chiba, "50-inch autostereoscopic full-clor 3-D TV display system", Proc. SPIE Vol 1669, Stereoscopic Displays and Applications III, June 1992

[19] H.Isono et al, "50-inch autostereoscopic 3-D television", *J.Inst. TV Engrs of Japan* Vol. 45, No. 11, pp. 1472-1474, 1991

[20] Y.Okita et al, "A 1.5-Megapixel a-Si TFT-LCD module for HDTV projector", SID 91 Digest, pp. 411-414, 1991

[21] K.Takeuchi et al, "A 750-TV-line-resolution projector using 1.5-Megapixel a-Si TFT LCD modules", SID 91, Digest, pp. 415-418, 1991

[22] D.Kahaner, US Office of Naval Research Asia – e-mail

[23] G.Lippmann, "Epreuves reversibles donnant la sensation du relief", *J. Phys. 7*, pp 821-825, 1908

[24] H.E.Ives, "Parallax panoramagrams made with a large diameter lens", *J. Opt. Soc. Amer.*, vol. 20, pp 332-342, 1930

[25] E.H.Adelson, J.Wang, "Single lens stereo with a plenoptic camera", *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol. 14, No. 2, Feb 1992

[26] D.Daly, R.F.Stevens, M.C.Hutley, N.Davies "The Manufacture of Microlenses by Melting Photoresist", *Meas.Sci.Technol.* 1 pp 759-766 (1990)

[27] R.F.Stevens, N.Davies, G.Milnthorpe "Lens Arrays and Optical System for Orthoscopic Three-Dimensional Imaging" *The Imaging Science Journal* Vol 49 pp 151-164 (2001)

[28] N.Davies, M.McCormick and L.Yang "Three-Dimensional Imaging System: a new development", *J.Appl. Opt.* 27. 4520-4528, (1988)

[29] F.Okano, H.Hoshino, J.Arai, I.Yuyama, "Real-time pickup method for a three-dimensional image based on integral photography", *Appl. Opt.* 36, pp. 1598-1603, 1997

[30] J.Arai, F.Okano, H.Hoshino, I.Yuyama, "Gradient-index lens-array method based on eal-time integral photography for three-dimensional images", *Appl. Opt.* 37, pp2034-2045, 1998

[31]B.Lee, S.Jung, S.Min, J.Park, "Three-dimensional display by use of integral photography with dynamic variable image planes", *Optics Letters*, Vol.26 No.19, Oct. 2001

[32] B.Lee, S.Min, B.Javidi, "Theoretical analysis for three-dimensional integral imaging systems with double devices", *Applied Optics*, Vol.41 No.23, Aug 2002

[33] S.Min, S.Jung, J.Park, B.Lee, "Study for wide-viewing integral photography using aspheric Fresnel-lens array", *Opt.Eng.* 41(10) 2572-2576, Oct 2002

[34] S.Min, S.Jung, J.Park, B.Lee, "Computer-generated integral photography", *6th International Workshop on 3-D Imaging Media Technology*,

[35] M.C.Forman, N.Davies, M.McCormick, "Objective quality measurement of integral 3D images", *SPIE Electronic Imaging : Stereoscopic Displays and Virtual Reality Systems IX*, San Jose, CA., USA, 2002

[36] M.C.Forman, N.Davies, M.McCormick, "Continuous parallax in discrete pixelated integral three-dimensional displays", *J.Opt. Soc. Am.* (accepted for publication), 2003

[37] Y.Igarashi, H.Murata, M.Ueda, "3D display system using a computer generated integral photograph", *Japanese Journal. Appl. Phys.* Vol.17 No.9, 1978

[38] A.Chutjian, R.Collier, "Recording and reconstructing three-dimensional images of computer generated subjects by Lippmann integral photography", *Appl. Opt.* 7 no.1, pp.99-101, 1968

[39] P.Cartwright, "Realisation of computer generated integral three dimensional images", *PhD Thesis*, Dec 2000

[40] J.K.Brown, M.McCormick, N.Davies, M.C.Forman, G.Milnthorpe, R.Kotecha, "The use of computer generated integral images to visualise cyber-structure", *20th Eurographics UK Conference*, Leicester UK, 2002

[41] J.Ren, A.Aggoun and M.McCormick, "Integration of virtual and real scenes *Systems for Optical Information Processing VI*, Seattle, WA., USA. 2002

[42] O.Youssef, A.Aggoun, W.Wolf and M.McCormick, "Pixels grouping and shadow cache for faster integral 3D ray tracing", *SPIE Electronic Imaging: Stereoscopic Displays and Virtual Reality Systems, IX*, San Jose, CA., USA 2002

[43] J.Whitted, "An improved illumination model for shaded display", *Comm. ACM*, Vol.23 No.6, 342-9, 1980

[44] A.Watt, "3D Computer Graphics", *Addison-Wesley Publishers Ltd*, 1993

[45] J.Blinn, "Models of light reflection for computer synthesized pictures", *Computer Graphics*, Vol.11 No.2, 192-8, 1977

[46] H.Gouraud, "Computer display of curved surfaces", *IEEE Trans.* 20, 623-8, 1971

[47] Bui-Tuong Phong, "Illumination for computer generated images", *CACM*, 18, 311-7, 1975

[48] W.Bouknight, "A procedure for the generation of three-dimensional half-toned computer graphics presentations", *Comm. ACM*, Vol.13 No.9, 527-536, 1970

[49] C.Wylie, G.Romney, D.Evans, A.Erdahl, "Halftone perspective drawings by computer", *Proc. AFIPS, Fall Joint Computer Conf.*, 31, 49-58, 1967

[50] D.F.Rogers,J.A.Adams,"Mathematical Elements for Computer Graphics",*2nd Ed. McGraw-Hill*,1990

[51] Paul Cartwright, "Realisation of Computer Generated Integral Three Dimensional Images", *Thesis, De Montfort University, Leicester*, p64, 2000

[52] Steve Upstill, "The Renderman Companion", *Addison-Wesley, New York*, 1989 p185-186

[53] G.B.Airy, Trans.Camb.Phil.Soc,5(1835),283

[54] W.T.Welford, "Aberrations of the Symmetrical Optical System", Academic Press (1974) – p81

[55] J.Warren Blaker, "Geometric Optics - The Matrix Theory", Marcel Dekker, New York 1971 - p80

[56] Geometric Optics – The Matrix Theory – J.Warren.Blaker – Dekker New York, 1971

[57] Max Born and Emil Wolf, "Principles of Optics, Electromagnetic Theory of Propogation, Interference and Diffraction of Light", 4th ed. *Pergamon Press*, Bath (1970)

[58] Crow. F.C, "A comparison of anti-aliasing techniques", *IEEE Computer Graphics and Applications*, 1 (1), 40-8

[59] J.K.Brown, M.McCormick, N.Davies, M.Forman, G.Milnthorpe, R.Kotecha, "The use of computer generated integral imaging to visualise cyber-structure", *Proc. 20th Eurographics UK Conference*, 3-8, June 2002

[60] G.Milnthorpe, M.McCormick, A.Aggoun, N.Davies, M.Forman, "Computer generated content for 3D TV displays", *IBC 2002*, Amsterdam

[61] G.Milnthorpe, M.McCormick, N.Davies, "Computer modeling of lens arrays for integral image rendering", *Proc. 20th Eurographics UK Conference*, 136-141,June 2002

[62] N.Badler, "Virtual humans for animation and ergonomics and simulation", *IEEE Workshop on Non-Rigid and Articulated Motion*, Puerto Rico, June 1997

[63] N.Magnenat, Thalmann, "Digital actors for interactive television", *IEE Proceedings, Part2*, p1022-1035, July 1995

[64] M.Price, J.Chandaria, O.Grau, G.A.Thomas, D.Chatting, J.Thorne, G.Milnthorpe, P.Woodward, L.Bull, E-J.Ong, A.Hilton, J.Mitchelson, J.Starck, "Real-time production and delivery of 3D media", *IBC 2002*, Amsterdam

[65]

[66] D.J. Chatting, J.M. Thorne, " Designing user interaction for face tracking applications", *DSV-IS 2002*, June 2002.

[67] N. Pelechano, L. Bull, M. Slater, "Fast Collision Detection Between Cloth and a Deformable Human Body ", (submitted)

[68] J. Starck, A. Hilton, and J. Illingworth, "Human Shape Estimation in a Multi-Camera Studio", *12th British Machine Vision Conference (BMVC)*, Manchester, UK, Vol. 2, pp. 573-582, September 2001.

[69] VRML Humanoid Animation Working Group, . *H-Anim 1.1 Specification*, 1999

[70] M.McCormick, N.Davies, G.Milnthorpe, A.Aggoun, M.Forman, "Integral imaging as a modality for 3D TV and displays", *Proc SPIE Vol 4864 59, Boston*, 2002

[71] R.F.Stevens, N.Davies, G.Milnthorpe, "Lens arrays and optical system for orthoscopic three-dimensional imaging", *Imaging Science Journal* 49(3), pp.151-164, 2001

[72] M. C. Forman, A. Aggoun, "Compression of full parallax integral 3D-TV image data", *SPIE Electronic Imaging '97: Stereoscopic Displays and Applications VIII*, San Jose, CA., USA.vol. 3012, pp. 222-226.

[73] R. Zaharia, A. Aggoun, M. McCormick, "Adaptive 3D-DCT compression algorithm fro continuous parallax 3D integral imaging", *Signal Process: Image Commun.* 17(3), pp. 231-242.

[74] A. Aggoun, I. Jalloh "A new 2D DCT/IDCT architecture", Accepted for publication by *IEE Proc.: Computers and Dig. Tech.* 2003

[75] Circuit Assembly Corps, 18 Thomas Street, Irvine, CA 92618-2777, Dwg. No 100252 Rev. B, 2000

[76]

[77] Greg Humphreys , Mike Houston , Ren Ng , Randall Frank , Sean Ahern , Peter D. Kirchner , James T. Klosowski, "Chromium: a stream processing framework for interactive graphics on clusters of workstations", *Proc. SIGGRAPH 2002*, July 200

[78] Gordon Stoll, Mathew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton Taveira, Steven Hunt, Pat Hanrahan, "Lightning-2: A high-performance display subsystem for PC clusters", *Proc. SIGGRAPH 2001*, 141-148, Aug 2001

[79] Blaxxun interactive AG , 1998

[80] T.Naemura, T.Yoshida, H.Harashima, "3-D computer graphics based on integral photography", *Optics Express* vol. 8 No. 2 Feb 2001